Dynamic Memory Bandwidth Allocation for Real-Time GPU-Based SoC Platforms

by

Homa Aghilinasab

A thesis
presented to the University Of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

The work presented in this thesis is partially based upon and extends the work presented in the following submitted paper:

[4]: Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. "Dynamic Memory Bandwidth Allocation for Real-Time GPU-Based SoC Platforms." International Conference on Embedded Software (EMSOFT), 2020, submitted for publication.

Due to the relation with the published work, parts of this thesis contain significant material from [4], including Chapters 3-5 and Section 2.3. We would like to thank all the co-authors for their precious help in completing this research. In particular, we would like to thank Waqar Ali and Heechul Yun for their work on the modification of the BWLOCK++ Linux kernel module [1], and Rodolfo Pellizzoni for his help on the Worst-Case Execution Time (WCET) estimation and budget allocation algorithms.

Abstract

Heterogeneous SoC platforms, comprising both general purpose CPUs and accelerators such as a GPU, are becoming increasingly attractive for real-time and mixed-criticality systems to cope with the computational demand of data parallel applications. However, contention for access to shared main memory can lead to significant performance degradation on both CPU and GPU. Existing work has shown that memory bandwidth throttling is effective in protecting real-time applications from memory-intensive, best-effort ones; however, due to the inherent pessimism involved in worst-case execution time estimation, such approaches can unduly restrict the bandwidth available to best-effort applications. In this work, we propose a novel memory bandwidth allocation scheme where we dynamically monitor the progress of a real-time application and increase the bandwidth share of best-effort ones whenever it is safe to do so. Specifically, we demonstrate our approach by protecting a real-time GPU kernel from best-effort CPU tasks. Based on profiling information, we first build a worst case execution time estimation model for the GPU kernel. Using such model, we then show how to dynamically recompute on-line the maximum memory budget that can be allocated to best-effort tasks without exceeding the kernel's assigned execution budget. We implement our proposed technique on NVIDIA embedded SoC and demonstrate its effectiveness on a variety of GPU and CPU benchmarks.

Acknowledgements

I would like to thank my supervisor Rodolfo Pellizzoni for his dedication, encouragement and guidance. This research and thesis would not have been accomplished without his constant support. I sincerely thank my committee members, Professor Hiren Patel and Professor Nachiket Kapre for reviewing this thesis.

Dedication

To my beloved parents and husband, Zahra, Mohammadreza, and Reza.

Table of Contents

List of Figures

List of Tables

List of Acronyms

GPU Graphics Processing Unit

CPU Central Processing Unit

WCET Worst-Case Execution Time

PREM Predictable Execution Model

ACET Average-Case Execution Time

CFG Control Flow Graph

DMA Direct Memory Access

TFS Throttle Fair Scheduler

WCL Worst-Case Latency

TDMA Time-Division Multiple Access

SoC System-on-Chip

DSAs Domain-Specific Architectures

LLC Last-Level Cache

HwCS Hardware Context Switch

CTA Concurrent Thread Arrays

SM Streaming Multiprocessors

PEG-C Performance Enhancement Guaranteed Cache

OS Operating System

EVT Extreme Value Theory

CBS Constant Bandwidth Server

BE Best Effort

EDF Empirical Distribution Function

K-S Kolmogorov-Smirnov

Chapter 1

Introduction

1.1 Motivation

Safety-critical embedded systems, in different areas such as avionics, automotive systems, medical devices, etc., are increasingly adopting high-performance computing architectures to meet rapidly growing computational demands. To cope with the breakdown of Dennard's scaling, Domain-Specific Architectures (DSAs) [40] are required to meet demanding targets in terms of performance-per-watt and performance-per-area. Graphics Processing Unit (GPU) is an example of such architecture, specialized for massively parallel applications. Examples of embedded applications employing GPUs for safety-critical processes include sensor data processing in robotics and autonomous cars [24]. Embedded systems tend to be highly resource-constrained in terms of power, cost, area, and weight. For this reason, heterogeneous System-on-Chip (SoC) platforms, integrating both general purpose CPU cores as well as GPU cores on the same device, are becoming the preferred solutions thanks to their ability to combine high performance and efficiency [52].

It is important to notice that due to the higher degree of integration, many such systems are mixed-critical, allowing applications with different criticality levels (for example: ASIL in ISO 26262 [2], or DAL in DO-178C [41]) to coexist on the same platform. Due to safety and fault propagation considerations, we will specifically consider systems where processing elements are partitioned based on the criticality of the applications they service. Therefore, we will distinguish between critical cores, which run safety-critical tasks, and Best Effort (BE) cores, which are used to execute lower-criticality applications. Many safety-critical tasks exhibit real-time requirements, where the correctness of the application depends on its ability to provide results before a predetermined point in time (a deadline). Hard real-

time systems do not tolerate any deadline miss, as such occurrence could lead to damage or loss of life. In contrast, soft real-time systems allow for some deadline misses to occur, or tolerate a finite lateness with respect to the deadline. A schedulability analysis is employed off-line to guarantee that real-time tasks will complete by their deadline under all possible execution scenarios. Such analysis relies on the Worst-Case Execution Time (WCET) of tasks as input. The estimation of WCET has to be safe, which means it should be above or equal to any possible execution time, and should also be tight, that is close to the actual execution time.

Unfortunately, the assumption of a constant per-task WCET breaks down on modern multi-core architectures, making it difficult to provide real-time guarantees. The key issue is the presence of various hardware resources, like caches, main memory, and buses, which are shared among cores. Contention for access to such resources can remarkably alter the WCET, effectively making the WCET of a task dependent on all other tasks in the system. In turn, this makes certification of mixed-critical systems essentially impossible, since the dependency means that low-criticality software applications can affect high-criticality ones and should therefore be certified at the highest (and most expensive) level.

In this thesis, we are particularly concerned with interference between the GPU and CPU cores, which share an individual main memory module in integrated SoC platforms. The interference of memory-intensive tasks running on BE processing elements, with the execution of critical real-time tasks running on real-time processing elements in parallel, can cause main memory bandwidth contention. To demonstrate this problem, we have evaluated the effect of running memory-intensive synthetic CPU tasks on the performance of the GPU benchmark *histo* from the `Parboil` suit [59] on a NVIDIA Jetson TX2 board, an approach similar to [8]. For this evaluation, we first run the GPU benchmark alone and record solo execution statistics. We then repeat the experiment by increasing the number of CPU cores running interfering memory-intensive tasks from one to three to effectively show main memory bandwidth contention. Figure 1.1 illustrates the result of this experiment. As it can be seen, co-scheduling the memory-intensive tasks on the CPU cores significantly increases the execution time of the GPU benchmark *histo*, up to 3.3X. Hence, in spite of the fact that the Jetson TX2 platform offers plenty of raw performance, no timing guarantee can be provided for a real-time task executing the GPU benchmark, unless the number and characteristics of co-running CPU tasks are known.

The key to break the timing dependencies between tasks, and support independent certification of applications with different criticality levels, is to provide timing isolation between cores. Specifically, the idea is to partition available hardware resources among the cores, so that the WCET of each task can be computed based on the system-wide

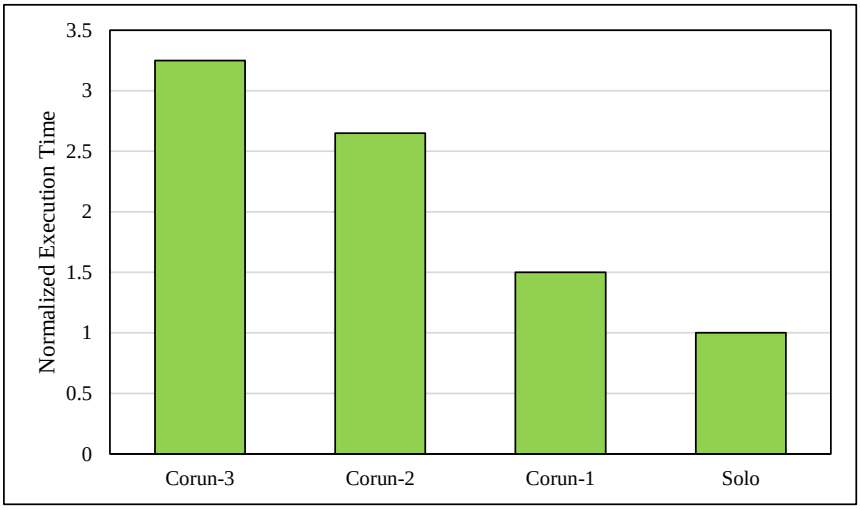


Figure 1.1: Performance reduction of histo benchmark due to increased number of interfering CPU cores

isolation parameters only, rather than the detailed behavior of co-running tasks. Since this thesis is concerned with main memory bandwidth contention, we specifically discuss memory regulation, a method by which we regulate a memory system to serve the requests of different requestors with defined rates. Memory regulation can be implemented at the hardware or software level. MemGuard [74] is one of the software methods to provide memory performance isolation in multi-core real-time systems. The goal is to ensure that the average memory access latency of a task is no larger than when executing on a system with a dedicated main memory that processes memory requests at a particular rate [74]. Hence, a multi-core system can be considered as a set of uni-core systems, each of which has a slower dedicated memory subsystem. Memguard bounds the bandwidth budget of each core by employing the hardware performance counters of the CPU cores. MemGuard assigns a budget, which is a predefined maximum access usage, to each core in every regulation period. If a core exceeds its budget, the core becomes idle until the next regulation period. As a result, the memory bandwidth is partitioned among cores guaranteeing a minimum bandwidth for each core. The same approach has been later ported to protect a real-time GPU kernel from interference generated by BE tasks executed on CPU cores in BWLOCK++[9].

While memory regulation can significantly reduce the maximum contention, and thus WCET inflation, suffered by a real-time GPU kernel, its impact on the performance of BE ap-

plications can be significant. For example, our evaluation in Section 5.4 shows that to limit WCET inflation to 10% for the memory-intensive kernel *histo*, we have to reduce the bandwidth of BE cores to 14% of their maximum throughput. While this severe constraint is needed to provide a WCET bound for the real-time kernel, it is important to notice that the contention caused by BE applications at run-time can sometimes be significantly less than the worst-case measurable one. In particular, computationally intensive BE applications might require low memory throughput and thus cause limited interference; furthermore, the pattern of memory accesses might be different from the worst-case one. Hence, by forcing a constant regulation budget based on the worst-case interference pattern, we might unnecessarily reduce the performance of BE tasks.

1.2 Methodology

To address the limitations of static memory bandwidth partitioning, in this work, we propose to adopt a dynamic approach to memory regulation. Specifically, when a GPU kernel first starts executing, our system enforces the statically-computed bandwidth budget for BE cores. We then monitor the progress of the kernel at run-time: if we determine that its execution is ahead compared to the worst-case behavior, we can increase the budget by dynamically re-computing the maximum BE bandwidth that allows the kernel to still complete within its original WCET; hence, increasing the performance of BE applications at no cost to real-time guarantees. In details, we target the following objectives:

1. Enforce main memory bandwidth isolation
2. Estimate the progress of the GPU kernel at run-time
3. Estimate the WCET of the GPU kernel
4. Re-compute the BE cores' bandwidth budget

Enforce main memory bandwidth isolation

We employ the existing BWLOCK++ framework to guarantee the worst-case memory bandwidth of the real-time processing element (the GPU) by regulating the number of injected requests from non-real-time ones (BE CPU cores). Some BE applications might require less bandwidth that the assigned regulation budget. Hence, the GPU kernel might execute faster compared to its worst-case behavior; this creates slack, the difference between

the WCET and the actual execution time of a real-time task. The idea is that, as the real-time GPU kernel accumulates slack at run-time, we can use the slack to increase the memory budget of BE cores, as long as the slow down suffered by the kernel for the rest of its execution is no larger than the accumulated slack. Note that we use the slack to improve the performance of BE applications, rather than other real-time tasks, because the real-time tasks fundamentally care about meeting deadlines, whereas BE applications care about average-case performance; and we already guarantee the real-time tasks' deadlines by computing their WCET while providing memory bandwidth isolation.

Estimate the progress of the GPU kernel at run-time

We propose a methodology to estimate the progress of a GPU kernel at run-time. Our methodology is based on the observation that a kernel executes a large number of threads with the same code; while such code can include control instructions, the number of different program paths is usually limited. We thus classify groups of threads into clusters, each with different execution time profiles; then, at run-time, we count the number of completed groups for each cluster as a measure of progress. We show how to implement the required instrumentation using NVIDIA CUDA programming framework and implement it on a Jetson TX2 board.

Estimate the WCET of the GPU kernel

Following the proposed progress mechanism, we introduce a measurement-based WCET approach to estimate the execution time of a kernel, based on its remaining number of thread groups per cluster, and the bandwidth budget for BE cores. Our WCET estimation method follows a hybrid approach: the execution time for each cluster of thread groups is first obtained by extensive measures under both isolation, and maximal interference by memory-intesive BE tasks. Then, the overall WCET of the kernel is analytically derived using the cluster information and the amount of time that BE tasks can perform memory requests based on the BWLOCK++ bandwidth budget.

Re-compute the BE cores' bandwidth

Finally, we propose an on-line dynamic budget allocation mechanism which re-computes the BE cores' bandwidth at each regulation period while ensuring that the kernel completes within its original WCET. We can guarantee that WCET of the GPU tasks will not

be exceeded, while the BE tasks can benefit from higher main memory bandwidth. We propose three different allocation schemes, which provide different grades of allocation fairness to BE tasks. We show that the approach can be implemented with low overhead on the Jetson TX2, and can lead to significant performance improvements using realistic benchmarks.

1.3 Thesis Outline

The rest of this thesis is organized as follows: in Chapter 2, we provide the background required to understand our approach and discuss related work. In Chapter 3, we introduce our system model and evaluation platform. Chapter 4 presents the clustering method and WCET estimation approach. In Chapter 5 we demonstrate the budget re-computation mechanism. Finally, we summarize the main contributions of this thesis and possible future work in Chapter 6.

Chapter 2

Background and Related Work

In this chapter, we present essential background for the work performed in this thesis and review related work. We begin by discussing work related to resource isolation in real-time systems, with a focus on homogeneous CPU-based systems, in Section 2.1. We then present background on Worst-Case Execution Time (WCET) analysis for real-time tasks in Section 2.2. Finally, we present background on GPU's execution model, and related work on real-time computing on GPU, in Section 2.3.

2.1 Resource Isolation in Real-Time Systems

Contention for access to shared hardware resources makes WCET estimation considerably more challenging. In this section, we discuss approaches that attempt to mitigate the effect of resource contention and provide timing isolation between CPU cores. The proposed methods consider different types of shared resources, such as cache and main memory. In this work, we focus on providing isolation at the level of the main memory bandwidth. From a performance perspective, contention at the level of shared caches can have an even more significant impact than contention in main memory; hence, we also discuss works targeting cache-level isolation. We categorize related works based on whether the mechanism is implemented in software only, or requires hardware modifications. Note that related work on memory regulation (specifically, Memguard [74]) have already been covered in Chapter 1.

2.1.1 Software Solutions

Contention in shared memory among multiple processing elements can lead to a very high increase in memory access latency [44, 63, 35]. The Predictable Execution Model (PREM), first proposed in [53], introduces a method for prevent contention in the shared memory of multi-core platforms. In particular, the authors divide programs into two phases: first, the memory phase, which is sensitive to contention, and second, the computation phase, which is free of contention. PREM schedules these phases such that two memory phases can never be executed in parallel. A task does not need access to the main memory during its computation phase, thereby allowing other processing elements to access it. By this rule, contention in memory can be avoided by construction. PREM effectively bounds memory access latency and provides a shorter WCET and improved hardware efficiency.

The approach has been refined in successive works [6, 66] into three phases. Specifically, two memory phases are considered: an acquisition (or load) phase that copies data and instructions from main memory into local memory, and a replication (or unload) phase that copies modified data back to main memory. While the computation phase is always executed on a processor, the memory phases can be either executed on the processor itself [5, 6, 13, 22, 26, 49, 50, 53, 56, 71, 72], or on another hardware component [30, 31], such as a programmable Direct Memory Access (DMA) module [7, 20, 61, 66]. Works that proposed using a DMA unit to perform the memory transfers [66] can efficiently hide the memory latency by overlapping the execution of a task with the DMA transfer of another task; this leads to considerable improvements in schedulability.

Other works provide isolation at the level of the shared cache through cache partitioning. Software partitioning techniques usually rely on indirect control over the cache, manipulating address-to-line mapping at compiler, Operating System (OS), or application-level [48, 23, 51]. Cache partitioning means assigning a given portion of the cache to a given task or core in the system to reduce space contention. The most common software-based cache partitioning technique is page coloring [47, 62]. Page coloring explores the virtual to physical page address translation present in the virtual memory system at the OS level when caches are physically indexed. Page addresses are mapped to predefined cache regions, avoiding the overlap of cache spaces.

Mancuso *et al.* proposed a cache allocation technique at the level of the OS kernel to improve performance with guaranteed predictable timing [48]. In this work, first, real-time tasks are profiled to determine the most accessed memory locations for each task. The profiling data is then used, at run-time, in a cache coloring and locking mechanism to help tighten the WCET for real-time tasks. Such a technique can be implemented regardless of the cache replacement policy and without any hardware modification. Ward *et al.* proposed

MC^2, a cache management strategy [65]. The authors used the page coloring mechanism along with cache scheduling instead of partitioning the cache statically. In MC^2, memory colors are treated as shared resources to which accesses must be arbitrated by either a real-time locking protocol or a scheduling algorithm.

A cache coherence protocol ensures the correctness of shared data across all cores in a multi-core platform. Nevertheless, adopting a conventional coherence protocol can reduce the predictability of the system: the latency suffered by one core accessing a shared line is dependent on the coherence state of that line in the private caches of other cores. Cache coherency can be implemented by employing software and hardware techniques. Software approaches for cache coherence require changing the application to handle the different copies of shared data explicitly. For instance, [55] modified the application to protect accesses to shared data by using lock mechanisms such that only one core at any time has access to the shared data. In the worst-case, this approach performs as well as the sequential execution of tasks sharing data.

2.1.2 Hardware Solutions

Several works have proposed to design new multi-core architectures to implement deterministic resource sharing schemes that provide better guarantees on the WCET of real-time tasks. Recently, the research community has introduced various predictable DRAM controller designs that provide improved worst-case latency for access to main memory. The proposed controllers are significantly different in terms of configuration, arbitration, etc. The authors in [32] presented a comprehensive evaluation for predictable DRAM controllers and attempted to classify the available controllers, and introduce an analytical performance model based on Worst-Case Latency (WCL).

Other works target cache modifications. Performance Enhancement Guaranteed Cache (PEG-C) [74] is a hardware addition to regular instruction cache in the form of a benefit counter for the hit and miss rates. The benefit counter follows the number of misses and at run-time provides access to the cache only when the value of the benefit counter is positive; otherwise, the access is served from memory. This hardware design not only addresses the unpredictability in the access to caches but also enhances the average performance compared to a regular cache.

Hardware Context Switch (HwCS) [10] is a hardware component which replaces the standard L1 cache controller of a processor. HwCS partitions the cache into two interchangeable layers. Each cache layer can either save in or load from the main memory,

while another layer is used as a usual L1 cache by the processor. HwCS makes the preemption overheads smaller compared to the task WCET since the cache content is saved after preemption and restored before resuming the task. Since both layers can access main memory simultaneously, memory bandwidth is divided between the two layers in the worst-case.

Hassan *et al.* [34] proposed PMSI, which is a predictable cache coherence protocol for multi-core systems. PMSI is the extended version of the classic MSI protocol and improved it with transient coherence states to bound the worst-case access latency. The implementation of PMSI only needs hardware modifications, and no changes to the OS and application. PMSI allowed tasks to access copies of shared data cached in their private caches simultaneously, which results in improved Average-Case Execution Time (ACET). Nevertheless, PMSI was not designed for mixed-criticality systems.

HourGlass [58] is a time-based predictable cache coherence protocol that is criticality-aware. HourGlass is designed specifically for dual-critical multi-core systems with only two criticality levels. This method ensures WCL bounds for memory requests originating from critical cores. By employing a timer-based mechanism, the bandwidth utilization of non-critical cores is improved while loosening the WCL bounds of critical tasks, which is acceptable until they meet their temporal requirements.

2.2 WCET Analysis

WCET analysis plays the important role of estimating an upper bound on the execution time of every real-time program that runs on a processing element. In classic real-time theory, schedulability analysis is based on the assumption of a fixed, known WCET for each task. Typically, the WCET of a task can be obtained in two ways:

Static Analysis employs the code of the real-time program in conjunction with a detailed model of the processing element to derive a safe WCET bound. This analysis method does not require the task to be actually executed on the hardware. It takes the code as input and finds the set of all possible Control Flow Graph (CFG) paths, and estimates an upper bound of WCET by determining the longest path based on the considered hardware model. The complex nature of multi-core and heterogeneous systems makes static analysis difficult or even impossible due to lack of detailed knowledge about the hardware behavior.

Measurement estimates the WCET by either running the program in a simulator or by executing it on the actual platform. This method measures the WCET by trying

different sets of inputs, thereby producing a range of execution times. Measurement-based methods are more suitable for soft real-time systems as they produce estimations rather than provable upper bounds. This method cannot guarantee to cover all possible execution paths; hence, the estimated WCET can be less than the actual WCET.

Bernat *et al.* [16] presented a technique to handle the interaction of complex hardware features by proposing probabilistic WCET estimation. The authors combine both analytical and measurement methods into a model for estimating probabilistically bounds on the execution time of the worst path of code sections. This idea is based on combining the worst-case effects seen in individual blocks to build the execution time model of the worst-case path of the program. Each basic block is given a probabilistic execution time distribution indicating that it is not always executed in the worst-case manner. Combining several basic block distributions constructs an execution time distribution for the whole program.

2.2.1 WCET Analysis in Multi-Core Systems

In multi-core systems, inter-core interference makes WCET analysis more complex compared to single-core platforms. In general, a bound on multi-core systems can be determined in two ways:

- Performing a joint analysis of all tasks and cores of the system. This way, the scheduling of the tasks and their allocation to the cores is known to the micro-architectural analysis. While this type of analysis may produce the most precise results, it is often disregarded due to the high computational complexity, rendering this approach infeasible [67].
- Performing separate WCET analyses for each task on each core, ignoring all interferences from the outside. Later, in a second step, the costs due to the interferences are analysed and incorporated into the results from the former analyses. Albeit computationally easier, this approach must be applied carefully due to the many non-timing-compositional features of modern processor architectures [67].

Few works have been proposed to address the challenge of analyzing shared caches in multi-core systems. These techniques are applicable for simple architectures and statically scheduled tasks. The first work that presents an analysis of shared caches in multi-core

systems is [70]. This work considers two tasks simultaneously executing on two cores with direct-mapped shared instruction cache. Later, cache conflict graphs were used for capturing the inter-core conflicts [75]. The work in [46] improves upon [70] by exploiting the lifetime information of tasks and excluding tasks that cannot overlap at run-time from the analysis. This work assumes all tasks are synchronized. Obviously, in any systems employing dynamic scheduling, it will be extremely hard to identify disjoint tasks. Other work [33] proposes to bypass the shared cache for single-usage cache lines to avoid inter-core conflicts and therefore improve the timing analysis.

Several works have considered the issue of computing WCL bounds for access to shared main memory in multi-core systems. The authors of [54] propose a framework for WCET analysis, which computes memory delay bounds for systems containing any number of cores and any number of peripheral buses with a single shared main memory. The authors provide two main contributions: first, they introduce the idea of computing a memory traffic arrival curve for each core, given a set of executed tasks. This curve presents an upper bound to the amount of memory traffic generated by the core in any time interval. Second, they describe an algorithm that computes a delay bound for a task given traffic curves for all other cores and peripheral buses in the system. The work in [44] considers the timing characteristics of resources in the main memory system. Their technique combines a request-driven approach that concentrates on the task memory requests, and a job-driven approach that focuses on interfering memory requests during task execution. The combination of both approaches allows the derivation of a tighter upper bound on the WCET of a task in the presence of memory interference.

2.3 GPUs in Real-Time Systems

A GPU is a highly parallel co-processor that performs operations requested by CPU code. A CPU application makes use of the GPU through a parallel-programming framework such as NVIDIA's CUDA, which offers standard APIs. A request to GPU typically comprises the following steps: 1) allocate memory in GPU's memory region and copy data from CPU memory to GPU memory; 2) launch the GPU function—called *kernel*—to process data in GPU memory; 3) wait for kernel completion; 4) copy the processed data from GPU memory region to CPU memory and 5) free the allocated GPU memory.

A GPU kernel consists of a combination of instruction code and a group of threads which execute those instructions. In CUDA terminology, this group of threads is denoted as a *thread block*; the number of thread blocks comprising the kernel and the dimensions of each thread block are specified by the programmer as part of the kernel's launch parameters. At

the hardware level, each thread block is processed by a number of hardware threads which form a *warp*. In NVIDIA GPUs, a warp comprises 32 hardware threads—executing in lock-step—and a number of warps can execute simultaneously on a streaming-multiprocessor (SM). A GPU consists of one or more SMs. For example, the integrated GPU in NVIDIA's Jetson TX-2 contains two SMs, each of which comprises 128 GPU cores and can thus execute up to 4 warps simultaneously. Overall, the GPU contains 256 cores and can execute instructions of up to 8 warps at any given time.

2.3.1 Thread Scheduling in CUDA

We next discuss in more details how thread scheduling is performed inside the GPU, as it affects our proposed solution in Chapters 4 and 5. Internally, the GPU contains a hardware scheduler which decides which warps to execute out of a pool of active warps. The behavior of the warp scheduler is proprietary and undisclosed; hence, we do not make any specific assumption on how warps are selected for execution. The pool of active warps is formed by selecting threads within a set of active thread blocks; within each thread block, threads are selected in increasing ID order. The number of active blocks depends on the GPU architecture and the resources consumed by each specific kernel; we will use M to denote the number of active blocks for a given kernel in the whole GPU (hence, for a GPU with two SMs, each SM is allocated $M/2$ blocks). When a kernel starts, blocks with IDs 0 to $M-1$ first become active; once all warps within a thread block complete execution, the block finishes and the not-yet started block with lowest ID becomes active. Hence, from the point of view of block scheduling, the GPU can be abstracted as a multiprocessor with M processors using a non-preemptive, global FIFO scheduling policy.

2.3.2 Real-Time Frameworks for GPU

Due to increased interest in GPU for accelerating parallel real-time applications, many real-time scheduling frameworks for GPU have been proposed in recent years [27, 45, 21, 37], with a particular focus on DNN acceleration [76, 69]. We first review works concerned with kernel scheduling, leaving more directly-related frameworks focusing on memory management to Section 2.3.3. Kato *et al.* proposed TimeGraph and RGEM frameworks that combine OS scheduling support for GPUs with timing guarantees [43, 42]. TimeGraph supports priority-based GPU scheduling to ensure hardware isolation of time-critical GPU tasks [43]. RGEM is a user-space approach that supports real-time GPU scheduling by ensuring the highest priority tasks are assigned to the GPU first [42]. RGEM

also breaks long copy operations into chunks that can be preempted. Elliott *et al.* proposed GPUSync [27] that provides real-time scheduling across one or more GPUs and supports sophisticated operations like copying data from one GPU to another. GPUSync has been used as a platform for real-time vision applications like those required in autonomous vehicles [28]. GPES is another software framework designed to support real-time processing on GPUs by making long-running kernels preemptible [77].

The most related work in this area is Merlot [57]. Similarly to our work, the authors of [57] note that WCET estimates for GPU kernels typically needs to be conservative. Therefore, they propose to monitor the execution of the kernel at run-time. If the kernel accumulates slack (i.e., its execution time is shorter than the worst-case), then such slack can be used to reduce the amount of resources used by the GPU and improve system performance. There are, however, three fundamental differences between Merlot and our approach. First of all, the goal of Merlot is to minimize the system's energy consumption, while our goal is to improve the memory throughput available to best-effort applications. Second, Merlot computes slack by dividing the kernel into a sequence of intervals and storing timing information for each interval. However, such approach does not work well with our memory regulation framework; instead, we show that we can perform run-time WCET estimation at the finer level of thread blocks. Finally, Merlot requires hardware modifications to the GPU, while our approach is software-based and can be implemented on available commercial platforms.

2.3.3 Memory-Aware Frameworks on GPU

In an integrated CPU-GPU platform, CPU and GPU share the same memory subsystem, which makes bandwidth sensitive GPU kernels susceptible to interference from CPU applications [31]. BWLOCK++ [9] is a software framework to protect GPU kernels from CPU-side interference on integrated CPU-GPU platforms. In BWLOCK++, one CPU core is dedicated to execute GPU using real-time tasks while the rest of the CPU cores are dedicated to execute best-effort CPU tasks. A GPU-using real-time task declares an acceptable interference budget from co-executing best-effort CPU tasks in the form of total number K of Last-Level Cache (LLC) miss events (and hence, fetches from main memory) from co-executing tasks that the subject task can tolerate in an interval of time T, called a *regulation period*. The budget K is split equally among the regulated CPU cores. A kernel level memory throttling framework [74] then limits the interfering memory traffic from co-executing CPU applications to the specified threshold value through periodic regulation using hardware performance monitoring counters. The implementation in [74, 9]

uses a value of T equal to 1-msec. In addition, BWLOCK+++ implements a throttling-aware best-effort CPU scheduling algorithm, called TFS, which favors CPU intensive tasks over memory intensive ones to minimize throttling while real-time GPU tasks are being executed. As we discuss in Chapter 3, our work used BWLOCK++ to enforce memory regulation for best-effort CPU cores.

In [39], the authors show how to partition GPU memory resources, including cache and main memory, to enforce strong isolation between concurrent kernels. However, the approach is highly platform-specific, requiring a great deal of reverse engineering, it is focused on discreet GPUs rather than integrated CPU-GPU SoCs, and does not protect the GPU from CPU interference.

A compiler-based technique to make GPU code PREM-compliant is introduced in [30]. Under PREM [53], each task has distinct computation and memory phases. The approach in [31, 30] ensures that the CPU does not perform memory accesses during the GPU memory phases, therefore eliminating memory contention by construction. However, it needs significant code restructuring, and can suffer significant overhead from the required fine-grained CPU-GPU synchronization.

In SiGAMMA [22], the authors introduced a mechanism for preempting the Graphics Processing Unit (GPU) kernel to protect critical real-time Central Processing Unit (CPU) applications. SiGAMMA is a server-based mechanism that operates as a memory arbiter between the CPU and GPU, to moderate the penalties arising with concurrent memory access by the GPU. CPU tasks follow PREM and have memory phases that have been separated from purely computation phases. A high priority spin kernel is used to preempt the currently running GPU kernel while a CPU is in its memory phase, thus preventing memory interference. This work is orthogonal to ours as it solves the problem of protecting CPU tasks from GPU tasks while our work solves the problem of protecting GPU tasks from CPU tasks.

2.3.4 WCET Estimation for GPU

Due to its complexity, WCET analysis for GPU kernels has received less attention compared to CPU analysis. A static analysis approach is introduced in [38], but it assumes a specific behavior of the warp scheduler that is not respected by commercial systems. The approach in [14] also employs static analysis, but with more relaxed assumptions. However, it can not handle cache stalls, and thus cannot be used in our context. A robust measurement-based probabilistic timing analysis is introduced in [15]. Similar to the approach we employ in Chapter 4, WCET estimation is based on collecting a trace of independent measurements.

However, the approach in [15] is applied at the level of the whole kernel, and thus cannot be used to estimate run-time progress.

All in all, GPU WCET estimation is difficult due to the lack of information about the hardware. GPU manufacturers rarely reveal specific implementation details, which are necessary to build a static analysis model, to maintain their competitive edge. For example, the associativity and replacement policies of the cache, the pipeline depth, and how exactly threads are scheduled on NVIDIA GPUs all remain undisclosed. For this reason, our approach only uses minimal assumptions on how threads are scheduled, as discussed in Section 2.3.1. The most related work is the hybrid analysis approach introduced in [19]. Here, the authors collect measurement traces at the level of individual warps, and then analytically compose the traces to derive the WCET of the whole kernel. Our approach in Chapter 4 also uses a hybrid analysis, but we apply it at the coarser level of thread blocks, since we find that analyzing traces at the warp level induces too much overhead for run-time implementation.

Chapter 3

System Model and Evaluation Platform

We begin by detailing the system model and key assumptions of our work in Section 3.1. Then, in Section 3.2 we introduce our evaluation platform and show how it meets such assumptions.

3.1 System Model and Assumptions

We consider an integrated CPU-GPU platform, comprising a GPU and multiple CPU cores, all sharing the same main memory. One core is used to execute real-time tasks, while the remaining cores execute best-effort applications with no real-time constraints. Only real-time tasks can use the GPU, by invoking the execution of a GPU kernel and suspending on the real-time core until the kernel completes. We do not make any assumption on the execution model or scheduling policy for best-effort applications; i.e., a regulation-aware scheduler such as TFS (see Section 2.3.3) can still be used to improve throughput of best-effort tasks under throttling.

Real-Time Task Model: we assume that the real-time core executes a set of periodic or sporadic real-time tasks. Each task τ_i comprises an alternating sequence of one or more CPU segments and zero or more GPU segments. Each GPU segment comprises the execution of a kernel $\kappa_{i,j}$, as well as the required memory copy operations. We assume that GPU operations are executed non-preemptively; while kernel preemption can improve the responsiveness of GPU operations [68, 21], it can also incur overhead in terms of additional

memory operations. We further assume that only one GPU kernel is executed at a time. While recent work has shown that co-scheduling multiple kernels can improve GPU resource utilization [76, 39], it also complicates the issue of timing analysis. For this reason, we reserve such an extension to future work. The methodology presented in this work does not require any further assumption on how real-time tasks are scheduled; the work in [8] presents a schedulability analysis for the same task model described above, assuming that tasks are scheduled according to fixed-priority preemptive policy on the CPU, and that bounds on the length of each memory copy operations, each kernel execution and the total amount of CPU execution are known.

Regulation Model: each kernel $\kappa_{i,j}$ is protected by enforcing a maximum budget ratio Q for best-effort cores through BWLOCK++ [9]. Recall from Section 2.3.3 that BWLOCK++ allows best-effort cores to perform up to K memory requests every regulation period of size $T = 1$ ms. Let $BW^{\max}$ to denote the maximum cumulative memory throughput that can be generated by the best-effort cores in number of LLC misses per second; we will obtain the value of $BW^{\max}$ experimentally in Section 4.5. Then, a budget ratio of Q corresponds to a regulation budget of $K = BW^{\max} \cdot T \cdot Q$ LLC misses per period T. Note that $Q = 0$ corresponds to the case where the kernel runs in isolation (without interference from the CPU), while $Q = 1$ corresponds to the case where no regulation is applied (maximum CPU-caused interference). Our WCET estimation method in Chapter 4 can compute a bound on the WCET $G^e_{i,j}(Q)$ of $\kappa_{i,j}$ for any given value of Q. For each kernel $\kappa_{i,j}$, we define a *nominal* budget ratio $\overline{Q}_{i,j}$, such that BWLOCK++ uses $Q = \overline{Q}_{i,j}$ at the start of the kernel. We select the highest budget ratio such that the slowdown of the GPU kernel, computed based on the estimated WCETs $G^e_{i,j}(\overline{Q}_{i,j})$ and $G^e_{i,j}(0)$, is within an acceptable margin (e.g., $< 10\%$ in our evaluation). The overarching goal of our approach is to *increase the actual budget ratio Q used at run-time as much as possible, while guaranteeing that the execution time of the kernel does not exceed its nominal WCET $G^e_{i,j}(\overline{Q}_{i,j})$*. This guarantees that schedulability analysis can be based on the nominal WCET computed off-line.

Platform Requirements: we assume that the scheduling of thread blocks follows the rules discussed in Section 2.3.1. To extract detailed timing information on each block, we further assume that the platform provides the following three functionalities: 1) a cycle accurate timer that can be used to count the elapsed time on the GPU since the beginning of a kernel; 2) a way to synchronize said GPU timer with the timer of the real-time core; 3) a mechanism to determine the IDs of all co-running thread blocks on the GPU.

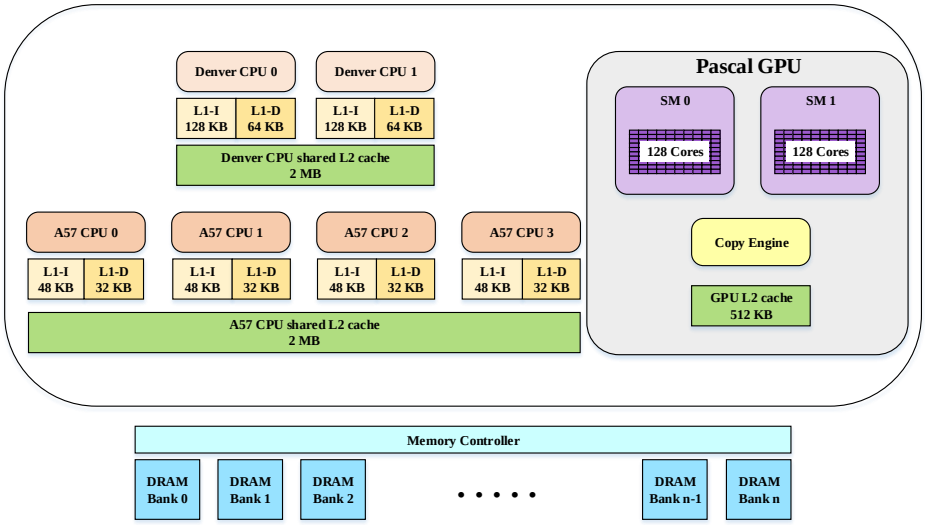


Figure 3.1: Jetson TX2 Architecture

3.2 Evaluation Platform

We use NVIDIA's Jetson TX-2 as our evaluation platform. As we can see in Figure 3.1 the Jetson TX-2 board contains a heterogeneous multi-core CPU cluster (4 Cortex A-57 + 2 Denver cores) and an integrated GPU. On the software side, we use NVIDIA's default Linux kernel (v4.4.38) and patch it with the changes required for memory bandwidth throttling of best-effort tasks through BWLOCK++ kernel module. As per our system model, we designate 3 Cortex-A57 cores as BE cores and one as the real-time CPU core; the real-time CPU core is not regulated whereas we use BWLOCK++ to regulate the LLC miss events (`L2_DCACHE_REFILL` in Cortex-A57 TRM [11]) of best-effort CPU cores. Please note that we only use the Cortex cores and disable the Denver cores in all our experiments because the latter lack support of necessary performance monitoring counters required by the memory throttling framework of BWLOCK++. Note that the Jetson TX-2 board meets all our platform requirements. Specifically, under CUDA the %ctaid registers can be read to determine the IDs of all running thread blocks, satisfying the third requirement. Furthermore, the GPU and CPU share a same clock timer, which can be used to satisfy the first two requirements.

Chapter 4

WCET Estimation

In this chapter, we introduce our first main contribution: a novel coarse-grained, hybrid Worst-Case Execution Time (WCET) estimation method for GPU kernels. We begin by introducing our hybrid method in Section 4.1. Since our methodology relies on measuring the execution time of thread blocks, in Section 4.2 we then discuss how blocks can be clustered into homogeneous groups based on their timing characteristics; then, Section 4.3 shows how to include the effects of memory regulation in the WCET estimation. Finally, Section 4.3 discusses the implementation of our methodology on our evaluation platform, specifically in terms of required kernel instrumentation, while we evaluate the results in Section 4.5 and conclude in Section 4.6

4.1 Hybrid WCET Estimation

Based on the discussion in Section 3.1, our goal is to estimate an upper bound on the completion time of a kernel based on the budget ratio Q assigned to BE cores. For simplicity of notation, in the rest of this section, we shall drop subscripts and use κ to refer to the kernel under analysis. As mentioned in Section 2.3.4, WCET estimation for GPU kernels is especially difficult because key architectural details, such as the way warps are scheduled, GPU caches are managed, etc., are both undisclosed and difficult to reverse-engineer. Inspired by the approach taken in [19], we thus propose to employ a hybrid approach to WCET analysis: specifically, we assume that the WCET of each thread block can be estimated through measurement-based techniques. We then analytically compose the per-block information to obtain a WCET bound for the whole kernel.

Hence, let N_κ denote the number of thread blocks for kernel κ, and $\forall i, 1 \leq i \leq N_\kappa$, let e_i denote the execution time of the i-th block. Without loss of generality, assume that the kernel starts at time 0, let j be the index of the block that finishes last in the kernel, and t_j be its starting time. Then by definition, the execution time of κ is equal to $t_j + e_j$. Recall from Section 2.3.1 that the GPU executes M thread blocks simultaneously. We thus note that since the j-th block is not started until time t_j, it follows that there must always be M other active thread blocks in the interval $[0, t_j)$. Therefore, it must hold: [1]

$$t_j \leq \Big(\sum_{i=1 \ldots N_\kappa} e_i - e_j \Big) / M; \tag{4.1}$$

and since $t_j + e_j$ is increasing in e_j, we obtain the following bound on the WCET G^e of the kernel:

$$G^e = \Big(\sum_{i=1 \ldots N_\kappa} e_i - e_{\max} \Big) / M + e_{\max}, \tag{4.2}$$

where $e_{\max} = \max_{i=1}^{N_\kappa} e_i$.

It remains to determine how to compute an upper bound to the execution time of each thread block e_i. Given that a kernel can comprise thousands of blocks, we find that maintaining per-block WCET information is too cumbersome, especially for on-line estimation. Instead, we propose to first classify the thread blocks in each kernel into clusters, where all blocks in the same cluster have similar execution profiles.

4.2 Block Clustering

All threads within the same kernel execute the same code; but due to the presence of control instructions such as branches and loops, different threads can execute along different code paths. We find that the following two observations typically hold for well-coded kernels: 1) the number of paths is small, as GPU code tends to be more predictable than CPU code. 2) It is highly desirable for threads within the same thread block to follow the same execution path: when threads in the same warp execute along different paths, the resulting thread divergence forces the GPU to execute the warp along all such paths, incurring a significant performance penalty. For these reasons, thread blocks can typically be classified into a small set of clusters.

[1] Note that our logic is equivalent to the computation of the interference rectangle in global scheduling analysis, see [17] for example.

Cluster #	1	2	3
worst-case measured e_i^0	1.70	2.50	3.30
worst-case measured e_i^1	3.69	6.52	8.83
$Q = 0$, mean	1.61	2.3	3.21
$Q = 0$, std	0.035	0.067	0.039
e_i^0 percentile	99.5%	99.9%	99.0%
$Q = 1$, mean	3.44	6.27	8.53
$Q = 1$, std	0.09	0.099	0.11
e_i^1 percentile	99.7%	99.4%	99.7%

Table 4.1: Clustering: hist benchmark. Time values are in us.

Our proposed clustering approach works in two steps: 1) first, we measure the execution time of each thread block by instrumenting the code of the kernel (see Section 4.4 for details) and executing it many times in isolation. This allows us to construct an Empirical Distribution Function (EDF) of the execution time of each block. While this process takes time and requires storing a large amount of data, we stress that the clustering step is performed off-line. 2) Based on the obtained EDFs, we then cluster thread blocks together by repeatedly applying the two-sample Kolmogorov-Smirnov (K-S) test [25] at a level $\alpha = 0.05$.

Let $\mathcal{C}$ be the resulting number of clusters, where the i-th cluster comprises N_i thread blocks. We can then modify Equation 4.2 to obtain an analytical WCET bound based on per-cluster, rather than per-block, execution time values. Specifically, let e_i^0 be the WCET for blocks of cluster i when executed in isolation (that is, with BE budget $Q = 0$), and let e_i^1 be the WCET when executed under maximum interference ($Q = 1$). We then obtain:

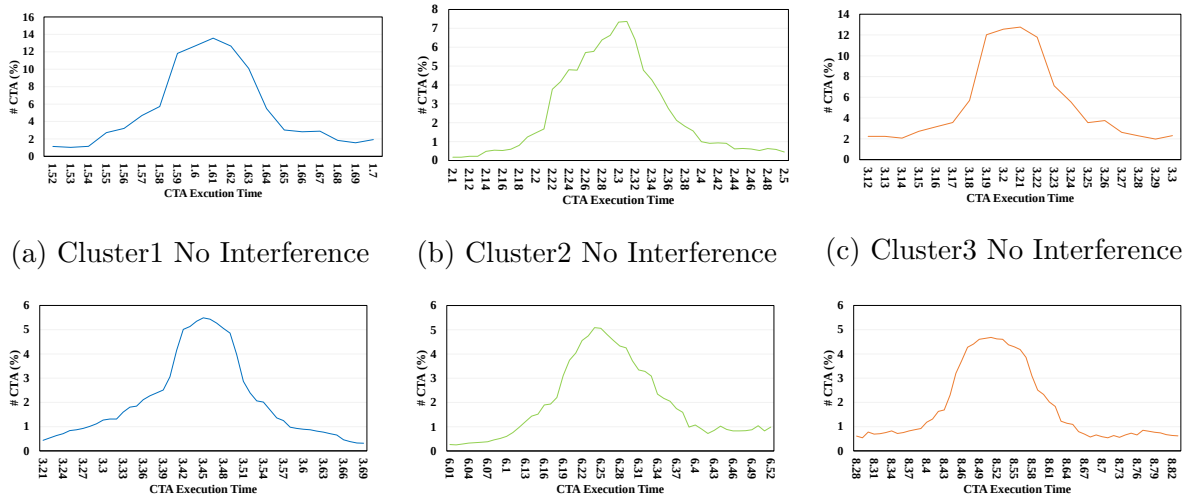
$$G^e(0, \{N_i\}) = \Big(\sum_{i=1...\mathcal{C}} N_i \cdot e_i^0 - e_{\max}^0 \Big) / M + e_{\max}^0, \tag{4.3}$$

for the WCET in isolation, and

$$G^e(1, \{N_i\}) = \Big(\sum_{i=1...\mathcal{C}} N_i \cdot e_i^1 - e_{\max}^1 \Big) / M + e_{\max}^1 \tag{4.4}$$

for the case $Q = 1$, where $e_{\max}^0, e_{\max}^1$ have the obvious meaning: $e_{\max}^0 = \max_{i=1}^{\mathcal{C}} e_i^0, e_{\max}^1 = \max_{i=1}^{\mathcal{C}} e_i^1$.

Finally, we point out that the problem of extracting the values of e_i^0 and e_i^1 from the EDF of each cluster is fundamentally orthogonal to our approach. In our evaluation,



(a) Cluster1 No Interference (b) Cluster2 No Interference (c) Cluster3 No Interference

(d) Cluster1 Full Interference (e) Cluster2 Full Interference (f) Cluster3 Full Interference

Figure 4.1: Distribution of block execution times for histo benchmark

we simply set them to the maximum observed execution time in the cluster. In Table 4.1, we report the corresponding values for the kernel of the *histo* benchmark [60], for which $\mathcal{C} = 3$; the kernel has been executed one million times with $Q = 0$ and one million times with $Q = 1$ and memory-intensive BE tasks. Following the approach in probabilistic timing analysis [15], we could otherwise fit the EDF to a test distribution, and then obtain e_i^0 (respectively, e_i^1) as a given percentile of the distribution, depending on the desired confidence level. To this end, we decided to again employ the K-S test with level $\alpha = 0.05$ for the fit of the execution time of each cluster to a normal distribution. As an example, we show detailed results for *histo* in Figure 4.1 and Table 4.1. Specifically, Figure 4.1 provides plots for the measured EDFs of all clusters, while Table 4.1 shows the obtained mean and standard deviation of the normal distribution for each cluster, the percentile level of the chosen e_i^0, e_i^1, and the goodness of fit, expressed as the ratio of the K-S statistic and the critical value of the K-S distribution [2]. We show the goodness of fit for other benchmarks in Section 4.5.3, while plots for the measured EDFs of other benchmarks are provided in appendix.

[2]The K-S statistic is the maximum difference between the EDF and the cumulative distribution function of the fit distribution; note that ratios below 1 indicate that the null hypothesis is not rejected, and hence the distributions are considered to be equal at the specified level.

4.3 Memory Interference Estimation

Equations 4.3 and 4.4 provide a way to compute the WCET of the kernel under either no interference ($Q = 0$) or full interference ($Q = 1$). It remains to determine how to bound the WCET for Q values between 0 and 1. This is significantly more difficult due to the way regulation works in our system: namely, BWLOCK++ does not mandate *when* BE cores can perform memory accesses during a regulation period, but only *how many* they can perform. Hence, without further assumptions on the interference model, we cannot determine the worst case memory request pattern. For this reason, we will provide a WCET estimation under the following *interference hypothesis*:

Hypothesis 1. *The interference suffered by a kernel for any value of Q is maximized when the BE cores issue requests at the same time and as fast as possible.*

We do not claim that Hypothesis 1 holds generally for all architectures and number of cores. Rather, in Section 4.5.2 we show through extensive testing that the hypothesis holds for our hardware platform; and we remark that systematic testing is accepted as proof of validation for even critical systems by certification authorities.

Under Hypothesis 1, in the worst case interference pattern the BE cores perform memory accesses at the maximum rate for $Q \cdot T$ time during each regulation period, and no memory access for the remaining $(1 - Q) \cdot T$ time. We call *memory time* and denote with t^{mem} the total amount of time, over the entire execution of the kernel, when BE cores perform memory accesses. We can then bound the WCET of κ by assuming that thread blocks which execute during the memory time take e_i^1 time to complete; while blocks which execute outside the memory time take e_i^0 each.

Algorithm 1 formalizes the corresponding WCET analysis. Note that the memory time depends on the number of regulation periods that the kernel's execution spans; while in turn, the WCET of the kernel depends on the memory time. To solve such circular dependency, Algorithm 1 iterates over the WCET t of the kernel, starting from the WCET in isolation $t = G^e(0, \{N_i\})$ (Equation 4.3). At each step, the algorithm first uses the value of t to determine the memory time, and then computes a new estimate for the WCET based on t^{mem}. The algorithm then set t to be equal to the new WCET bound and iterates until convergence.

Figure 4.2 shows how to compute t^{mem}. The worst case scenario changes whether the kernel starts at the same time as a regulation period ($sync = 1$), or no such assumption can be made ($sync = 0$). In the latter case, t^{mem} is maximized when the BE cores access memory as late as possible in the first interval, and as soon as possible in all other intervals;

Algorithm 1: WCET Estimation

Input: $Q, \{N_i\}, \{e_i^0\}, \{e_i^1\}, sync$
Output: Kernel WCET

```
t = G^e(0, {N_i})
while 1 do
    Compute t^mem(t, Q, sync) based on Eq. 4.5
    Compute G^e(Q, {N_i}, sync) by solving Eq. 4.8-4.10
    if t == G^e(Q, {N_i}, sync) then
        return t
    t = G^e(Q, {N_i}, sync)
end
```

and the beginning of the kernel is aligned with the start of the BE accesses. Based on such patterns, we obtain: [3]

$$t^{mem}(t, Q, sync) = \begin{cases} t & \text{if } t \leq t^{init}; \\ (1 - sync + P) \cdot Q \cdot T + & \\ \min(t - t^{init} - P \cdot T, Q \cdot T) & \text{otherwise,} \end{cases} \tag{4.5}$$

where:

$$t^{init} = (1 - sync) \cdot Q \cdot T, \tag{4.6}$$

$$P = \left\lfloor \frac{t - t^{init}}{T} \right\rfloor. \tag{4.7}$$

Finally, we consider $G^e(Q, \{N_i\}, sync)$. Let x_i, with $x_i \leq N_i$, to denote the number of blocks in the i-th cluster that execute during the memory time. We can then bound the WCET of the kernel by solving the following problem:

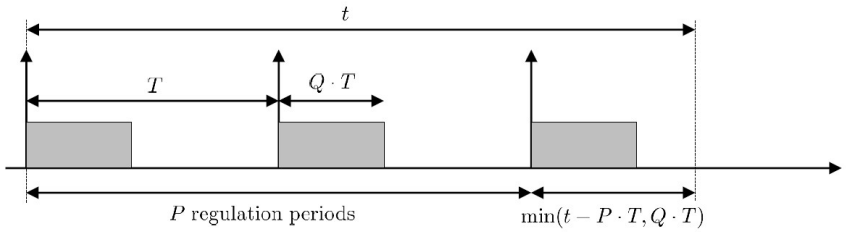
$$\max G^e(Q, \{N_i\}, sync) =$$

$$\Big(\sum_{i=1...\mathcal{C}} x_i \cdot e_i^1 + (N_i - x_i) \cdot e_i^0 - e_{\max}^1 \Big) / M + e_{\max}^1 \tag{4.8}$$

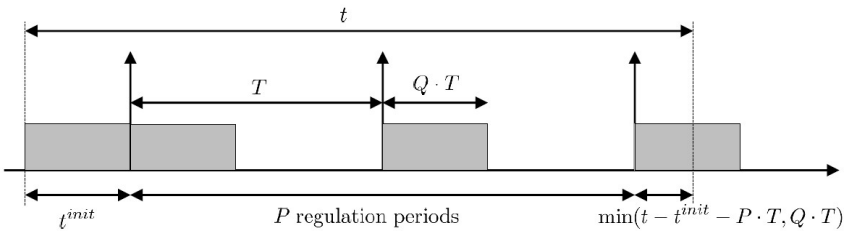
$$\Big(\sum_{i=1...\mathcal{C}} x_i \cdot e_i^1 \Big) / M \leq t^{mem}(t, Q, sync) \tag{4.9}$$

$$\forall i = 1...\mathcal{C} : 0 \leq x_i \leq N_i \tag{4.10}$$

[3] Note that the derivation is equivalent to computing the workload of a sporadic task in a problem window under global scheduling [17].



(a) $sync = 1$



(b) $sync = 0$

Figure 4.2: t^{mem} derivation

Note that here, Equation 4.9 constraints the variables $\{x_i\}$ based on the length of the memory time, while Equation 4.8 bounds the WCET in the same way as Equation 4.2. To solve the problem in polynomial time, we overapproximate the WCET bound by relaxing the variables $\{x_i\}$ to be real rather than integer. Under such relaxation, and since by definition $e_i^1 \geq e_i^0$, it is easy to see that the objective function in Equation 4.8 is maximized when Equation 4.9 holds with equality, i.e. $(\sum_{i=1}^{\mathcal{C}} x_i \cdot e_i^1)/M = t^{mem}(t, Q, sync)$. If we define $\forall i : y_i = x_i \cdot e_i^1$, the problem is then equivalent to:

$$\max CONST - \sum_{i=1...\mathcal{C}} y_i \cdot \frac{e_i^0}{e_i^1} \tag{4.11}$$

$$\sum_{i=1...\mathcal{C}} y_i = t^{mem}(t, Q, sync) \cdot M \tag{4.12}$$

$$\forall i = 1...\mathcal{C} : 0 \leq y_i \leq N_i \cdot e_i^1 \tag{4.13}$$

where $CONST$ does not depend on variables $\{y_i\}$ (it is constant). It is then obvious that the problem can be solved in the following greedy manner: 1) start with all variables $\{y_i\}$ (equivalently, $\{x_i\}$) set to zero; 2) order the clusters by increasing values of the ratio e_i^0/e_i^1 and select one cluster at a time in such order (i.e., select first the cluster which experiences the largest relative increase in execution, that is, the cluster which is most susceptible to

Algorithm 2: Measure Block Execution Time

```
1  if threadId.x == 0 then
2      clk = getclock( )
3      if blockId.x > M then
4          read co-running thread block IDs in IDlist
5          find i such that TimeAr[i].ID is not in IDlist
6          Duration = clk - TimerAr[i].clk
7          Write Duration to main memory array
8      else
9          i = blockId.x
10     TimeAr[i].ID = blockId.x
11     TimeAr[i].clk = clk
```

memory interference); 3) for the selected cluster, increase y_i (equivalently, x_i) until either Equation 4.12 or 4.13 saturates; 4) if 4.13 saturates before 4.12, go to the next cluster and repeat from step 2.

4.4 Implementation

We next explain how we instrument the code of a GPU kernel to measure the execution time of each thread block. As discussed in Section 2.3.1, whenever a block finishes, another block is assigned to the GPU for execution immediately; hence, the finish time of the terminated block is the start time of the newly assigned block. The execution time of the terminated block is then computed as the difference between finish and start time.

For ease of exposition, Algorithm 2 shows the pseudo-code of the instrumentation; complete CUDA code is provided in appendix. We allocate two data structures. An array of size N_κ in main memory is used by the GPU to store the computed execution time of each thread block; after the kernel finishes executing, we read the array content from the CPU and use it as input to the clustering process. TimeAr is an array of size M allocated in GPU memory; each element of the array is a structure comprising the ID and the start time of an active thread block. Based on Line 1, the instrumentation code is executed by the first thread of each block; since threads in each block are selected in order, this guarantees that the thread execution coincides with the start time of the block. The thread reads the current clock time at Line 2, as well as the list of co-running thread blocks at Line 4. The list of co-running blocks is then matched based on IDs to the content of

TimeAr to determine which thread block has finished (note this is done only after the first M blocks have started), and the execution time of the finished block is computed (Lines 5-7). Finally, the ID and start time of the new block is saved in TimeAr.

Note that this scheme cannot measure the execution time for the M thread blocks of the kernel that finish last, since no new block will start after their termination. Hence, we only use data for $N_\kappa - M$ blocks to perform the clustering. After clustering, we then rerun the kernel with a modified instrumentation, where we estimate the execution time of each of the last M thread blocks as the difference between the start time of the last and first thread in the block. While such measurement is much less precise than what obtained by Algorithm 2, we found it sufficient to classify each thread block in one of the previously obtained clusters.

4.5 Evaluation

In this section, we report results for our presented WCET estimation framework. Similarly to [9], we use six memory bandwidth sensitive GPU benchmarks from Parboil suite [60]. Table 4.2 details benchmarks' characteristics which will be discussed throughout the rest of this section. Note that each benchmark invokes the same kernel multiple times, possibly with a different input set size; for this reason, we decided to report information for the first kernel invocation in each benchmark.

4.5.1 Bandwidth and Budget Estimation

We employ bandwidth benchmark from IsolBench suite [63] as our synthetic memory-intensive CPU application to cause maximum interference on GPU kernels. The bandwidth benchmark linearly accesses a 1-D array of configurable size and the sequential write pattern of this benchmark is known to cause worst-case interference on several multi-core platforms [64]. In our case, we configure bandwidth benchmark to generate LLC misses; thus creating memory level interference. We also use this benchmark to determine the $BW^{\max}$ value for our TX-2 platform. For this purpose, we run three instances of bandwidth benchmark on the 3 Cortex-A57 cores. Our measurement shows that the maximum cumulative memory bandwidth of the three bandwidth benchmarks is ∼3.9 GB/s. In terms of LLC misses, this corresponds to ∼60,000 events per regulation interval of 1-msec. Divided over 3 best-effort CPU cores, this is equal to 20,000 LLC misses in each regulation interval which corresponds to $Q = 1$ i.e., maximum possible interference from each best-effort core.

Benchmark	histo	sad	bfs	spmv	stencil	lbm
Number of thread blocks	37,627	59,136	82,318	31,624	31,952	63,627
Number of clusters	3	6	6	3	4	5
Goodness of fit	0.76	0.45	0.53	0.68	0.86	0.66
Cluster intervals	14	21	29	17	19	25
Analytical WCET overestimation at $Q = 0$ (%)	5	9.1	0.6	5.6	5.2	2
Analytical WCET overestimation at $Q = 1$ (%)	23.2	26.9	11.3	9.1	7	25.7

Table 4.2: Benchmark Characterization

4.5.2 Testing the Interference Hypothesis

We validated Hypothesis 1 through extensive testing. Specifically, we synchronized the execution of three copies of the synthetic bandwidth benchmark running on the three BE cores, and modified the benchmark code to randomly vary: 1) the offsets, relative to the beginning of the regulation period, at which each BE core starts execution; 2) the ratio of read and write operations; 3) the time separation between LLC misses, controlled by inserting a variable number of NOP instructions between reads or writes. We then executed each kernel for several hours and recorded its worst case execution time. In all cases, we found that the WCET is maximized for a 100% write ratio with no NOP added and equal offsets for all cores, which matches the hypothesis.

4.5.3 Clustering Results

Table 4.2 shows the number of thread blocks per benchmark, as well as the clustering results, in terms of number of clusters, intervals, and the worst goodness of fit (i.e., the maximum ratio) over all clusters of each benchmark. We find that the numbers of both clusters and intervals is small for all benchmarks, leading to small memory space overhead for the data structures in Section 4.4, and low run-time for the on-line algorithm.

4.5.4 Tightness of WCET Estimation

To estimate the tightness of the hybrid WCET bounds, we run the following experiment: we first run each kernel one million time for varying values of Q, without any instrumentation and together with the synthetic bandwidth benchmark, and determine its worst-case measured execution time. To reduce variability, we further force the kernel to start synchronously with the regulation period, i.e. $sync = 1$. We then compare such measured

WCET with the analytical WCET obtained through Algorithm 1, and report in Table 4.2 the overestimation ratio at both $Q = 0$ and $Q = 1$ for all benchmarks; while Figure 4.3 shows the detailed WCET plots for all benchmarks as a function of Q. We point out that the overestimation is due to three factors: 1) the measured execution does not represent the real worst-case, as the bandwidth benchmark performs all memory requests at the beginning of each regulation period; hence, it might fail to align memory requests with the most interference-sensitive thread blocks. 2) Our clustering approach can lead to some over approximation of the real WCET of the thread block. However, we expect such effect to be limited, since as shown in Table 4.1, the fitted standard deviations for the clusters tend to be rather small. 3) Finally, the instrumentation adds some timing overhead; however, this is again small, at most 1.5% for the tested benchmarks, measured by running each benchmarks in isolation with and without instrumentation.

4.6 Discussion and Conclusions

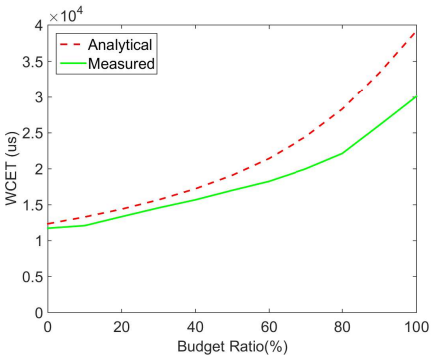
In this chapter, we presented our hybrid WCET estimation method. Compared to other techniques for WCET estimation of GPU kernels, our approach makes very limited assumptions on the internal behavior of the GPU, which is largely unknown for commercial platforms. Results show that our clustering technique is effective in grouping thread blocks with similar characteristics, leading to relatively limited pessimism in WCET estimation.

We make four additional, important observations. First, as pointed out in Section 4.2, the WCET results necessarily depend on our choice of selecting the values of e_i^0, e_i^1 as the worst-case observed values. Based on the percentiles reported in Table 4.1, we acknowledge that higher percentiles would be needed to satisfy strict certification requirements, and this would lead to more pessimistic WCET estimates. However, we point out that in this case, the performance of the dynamic allocator presented in Chapter 5 would actually *increase*, since a wider gap between estimated and actual WCET would provide a higher ability to reclaim memory budget for best-effort cores.

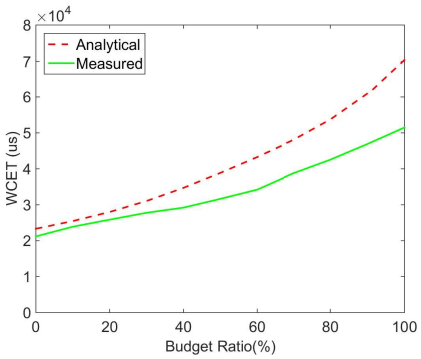
Second, since our WCET estimation depends on Hypothesis 1, we cannot claim generality beyond our tested platform. However, we believe it is likely that the hypothesis holds on other platforms as well, since several previous studies have highlighted that worst-case delays are generated when hardware request queues saturate [63, 12], and maximizing concurrent activity of all cores increases the probability of such occurrence. Furthermore, in case the hypothesis does not hold, but a precise model of main memory is available, we argue that a more complex analysis, along the lines of [35, 73], could be used to bound the maximum delay suffered by the kernel.

Third, our approach requires modifications to the code of each GPU kernel. For the sake of obtaining the results in this thesis, we modified each benchmark manually. We point out that the required modification is rather simple, limited to adding some variable declarations and code at the beginning of each kernel; hence, it is reasonable to assume that the approach could be automated at the compiler level, possibly by modifying executable code (CUDA PTX) so that source code access is not required.

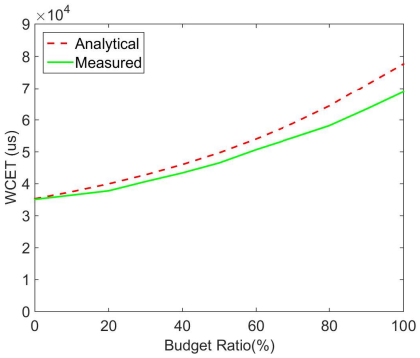
Finally, note that the complexity of our WCET estimation Algorithm 1 is pseudo-polynomial in the kernel's characteristics, since the algorithm is required to iterate over t. However, the algorithm is only used off-line: as we show in Chapter 5, our online budget allocation scheme employs a faster algorithm that has linear complexity in the number of clusters.



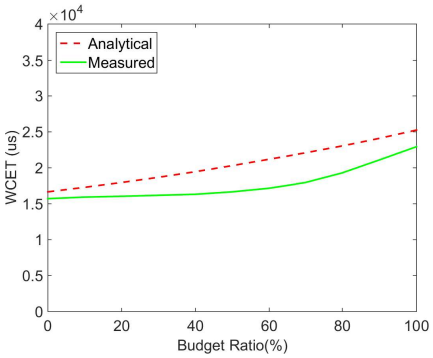
(a) histo



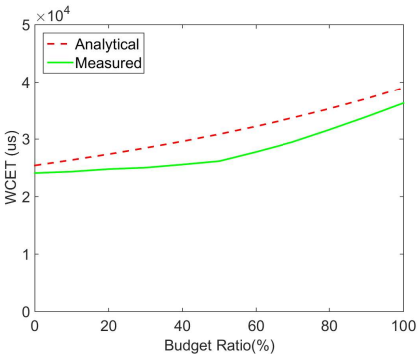
(b) sad



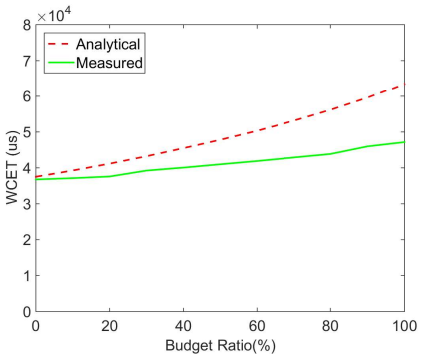
(c) bfs



(d) spmv



(e) stencil



(f) lbm

Figure 4.3: Analytical WCET vs Measured WCET for *histo*

Chapter 5

Dynamic Budget Allocation

In this chapter, we present the second main contribution of this thesis: a run-time algorithm to dynamically allocate the memory budget ratio Q to BE cores, while guaranteeing that the real-time GPU kernel completes within its original execution time, which is determined off-line based on a nominal memory budget. We begin by presenting the basic allocation scheme in Section 5.1, and then introduce alternative improved methods in Section 5.2. We elaborate on our implementation in Section 5.3; this includes modifications to the GPU kernel code, the existing BWLOCK++ OS-level implementation, and the introduction of a user-level process to perform the actual budget computation. Finally, we evaluate results in Section 5.4 and provide discussion and concluding remarks in Section 5.5.

5.1 Allocation Algorithm

As discussed in Section 3.1, let $\overline{Q}$ be the nominal budget for kernel under analysis κ. Since the kernel can be invoked at any point in time, we cannot guarantee that its start time $\bar{t}$ coincides with the beginning of a regulation period. Hence, following the analysis in Section 4.3, we have $sync = 0$ and we let $G^e(\overline{Q}, \{N_i\}, 0)$ be the computed WCET bound for κ. Then, the real-time requirement that our algorithm satisfies is to ensure that κ finishes no later than $\bar{t} + G^e(\overline{Q}, \{N_i\}, 0)$.

Specifically, we employ the following approach. The first regulation period, which starts before or at the beginning of the kernel, is assigned the nominal budget $\overline{Q}$. For each successive time t^{reg}, corresponding to the beginning of a regulation period, we perform two steps: 1) first, we determine the number of remaining thread blocks $\{R_i\}$ for each cluster.

Algorithm 3: On-line Budget Computation

Input: $t, \{R_i\}, \{e_i^0\}, \{e_i^1\}$, with clusters ordered by increasing e_i^0/e_i^1
Output: Budget ratio Q for next regulation period

1 **for** $j = 1...\mathcal{C}$ **do**
2 **if** $(\sum_{i=1...j} R_i \cdot e_i^1 + \sum_{i=j+1...\mathcal{C}} R_i \cdot e_i^0 - e_{\max}^1)/M + e_{\max}^1 \geq t$ **then**
3 $x_j = \frac{(t-e_{\max}^1)\cdot M - \sum_{i=1}^{j-1} R_i \cdot e_i^1 - \sum_{i=j}^{\mathcal{C}} R_i \cdot e_i^0 + e_{\max}^1}{e_j^1 - e_j^0}$
4 $t_{\max}^{mem} = (\sum_{i=1}^{j-1} R_i \cdot e_i^1 + x_j \cdot e_j^1)/M$
5 **return** max Q s.t. $t^{mem}(t, Q, 1) = t_{\max}^{mem}$ by inverting Equation 4.5
6 **end**
7 **return** 1

We instrument the code similarly to Section 4.2 to determine which blocks have finished, and use this information to determine R_i; details are provided in Section 5.3. 2) Based on the remaining thread blocks, we determine the maximum budget Q that can be assigned to the next regulation period while ensuring that κ completes by $\bar{t} + G^e(\overline{Q}, \{N_i\}, 0)$.

We next discuss the second step. A trivial solution would be to employ Algorithm 1; since we know that t^{reg} corresponds to the beginning of a regulation period, and the number of not-yet-completed blocks in each cluster is R_i, it follows that for a budget Q, the kernel must complete by $t^{reg} + G^e(Q, \{R_i\}, 1)$. We could thus use binary search to find the maximum Q such that $t^{reg} + G^e(Q, \{R_i\}, 1) \leq \bar{t} + G^e(\overline{Q}, \{N_i\}, 0)$, or equivalently:

$$\max_Q : G^e(Q, \{R_i\}, 1) \leq G^e(\overline{Q}, \{N_i\}, 0) - (t^{reg} - \bar{t}). \tag{5.1}$$

Note that in Equation 5.1, $G^e(\overline{Q}, \{N_i\}, 0)$ is a constant computed off-line. To speed up the on-line computation for $G^e(Q, \{R_i\}, 1)$, we next propose a faster algorithm. The key idea is that since we know that the remaining time to complete execution is $t = G^e(\overline{Q}, \{N_i\}, 0) - (t^{reg} - \bar{t})$, we do not need to iterate over time t as in Algorithm 1.

We show our solution in Algorithm 3. Following the same principle as in the greedy solution for the problem in Equations 4.11-4.13, the algorithm iterates over clusters starting from the most interference-sensitive one, and tries to maximize the number of blocks of that cluster that execute with maximum interference (i.e., for e_i^1 each during t^{mem}). The iteration stops when the condition at Line 2 is met; note that the right side of the condition computes the WCET for the kernel assuming that all blocks of clusters $1...j$ execute for e_i^1, while all other clusters execute for e_i^0. If the condition is never met, then the kernel can complete within the allocated time t even under maximum interference for all blocks,

hence we return $Q = 1$. Otherwise, let x_j to denote the maximum number of thread blocks of cluster j that can execute during t^{mem}; we can rewrite the equation at Line 2 to:

$$\Big(\sum_{i=1\ldots j-1} R_i \cdot e_i^1 + x_j \cdot e_j^1 + (R_j - x_j) \cdot e_j^0 + \sum_{i=j+1\ldots \mathcal{C}} R_i \cdot e_i^0 - e_{\max}^1 \Big) / M + e_{\max}^1 = t, \tag{5.2}$$

and solving the equation yields the assignment to x_j in Line 3 of the algorithm. Finally, we compute the value of t^{mem} at Line 4 (which is equivalent to Equation 4.9,4.12), and then determine the value of Q by using the inverse of the function in Equation 4.5.

5.2 Improved Allocation

It is interesting to note that the strategy presented in Algorithm 3, which we call the *FAIR* allocation, consists in splitting the memory time fairly among all remaining regulation periods: the same value of Q is used for all future periods. However, there is no requirement to allocate the budget in a fair manner: we could instead compute the worst-case finish time of the kernel by increasing only the budget of the next period, while keeping all other periods to the nominal budget $\overline{Q}$. The idea is that the kernel is likely to accumulate more slack while executing during the next regulation period, at the end of which we will run the allocation algorithm again to re-compute the budget Q; hence, such *GREEDY* allocation might result in higher budget values for the first regulation periods of the kernel execution.

Under *GREEDY*, the budget for the next regulation period is computed by first determining the maximum memory time during the next period: this is equal to t_{max}^{mem}, as computed at Line 4 of Algorithm 3, minus the memory time of all other periods under nominal budget, which is $\max\big(0, t^{mem}(\overline{Q}, t - T, 1)\big)$. Since the memory time in a single regulation period is by definition equal to $Q \cdot T$, we then obtain:

$$Q = \min\Big(1, \big(t_{max}^{mem} - \max(0, t^{mem}(\overline{Q}, t - T, 1))\big)/T\Big), \tag{5.3}$$

which replaces Line 5 in Algorithm 3. Finally, as we will show in Section 5.4, a downside of the *GREEDY* allocation is that the budget ratio Q can change widely between successive regulation periods, possibly leading to a rather unfair bandwidth allocation for co-running BE applications. We thus propose a third allocation scheme, which we call *SMOOTH*, which modifies the *GREEDY* allocation by applying a simple filter of the form:

$$y_n = \min\big(x_n, a \cdot x_n + (1 - a) \cdot y_{n-1}\big), \tag{5.4}$$

where x_n is the budget computed by *GREEDY* for the current regulation period and y_{n-1}, y_n are the budgets selected for the previous and current period, respectively. Based on our evaluation, we experimentally set a value $a = 0.3$ for the smoothing parameter.

5.3 Implementation

Because of the nuances involved in making necessary CUDA library calls from a Linux kernel module, we do not implement our on-line budget computation algorithm at the OS kernel level. Instead, we implement it in a user-level high priority real-time process which runs concurrently to the GPU kernels on the real-time CPU core. At the kernel level, we use BWLOCK++ Linux kernel module [1] to enforce the computed budget values for regulating the memory bandwidth of best-effort CPU cores. For this purpose, we implement a shared-memory based communication mechanism between the kernel module and the user-space budget-computation process. Concretely, for each regulation period, the user-space process calculates a new budget value as per Algorithm 3 which is then written to a predefined shared-memory area. The kernel module reads the new budget value from the shared-memory and enforces it in the current regulation interval. Note that the budget-computation process takes some time to perform the required computation and pass the information to BWLOCK++; for this reason, we synchronize it to start computation a fixed amount of time before the start of each regulation period of BWLOCK++. We note that the resulting budget computation is still safe albeit more pessimistic: the extra time might cause us to miss some completed thread blocks, but this would lead to higher values of $\{R_i\}$ and hence a higher WCET estimate and a lower computed Q. However, due to the pessimism, it is theoretically possible to compute a value of Q that is less than the nominal budget $\overline{Q}$, in which case we can still safely set $Q = \overline{Q}$.

Similarly to Section 4.4, we allocate a data structure in main memory that can be accessed by both the GPU and the user-level process. We instrument the kernel code so that it writes the current clock value at the beginning of the first thread block, which we take as the starting time $\bar{t}$ of the kernel itself; the budget-computation process uses the value passed by the GPU to compute the elapsed time $t^{reg} - \bar{t}$ for the kernel. Each successive thread block writes to main memory the IDlist of concurrent blocks, which is used by the budget-computation process to determine the remaining blocks $\{R_i\}$. Complete CUDA code for the instrumentation is provided in appendix.

To efficiently compute $\{R_i\}$, we use the concept of a *cluster interval* $[i, j]$, that is, a sequence of thread blocks where all blocks with IDs in $[i, j]$ belong to the same cluster. The budget-computation process is provided with a table, computed off-line, where each

Benchmark	histo	sad	bfs	spmv	stencil	lbm
Nominal budget (%)	14	12	17	28	28	22.5
Adjusted nominal budget (%)	13.2	11.2	16.2	27.2	27.2	21.7

Table 5.1: Nominal Budgets for all Benchmarks

Benchmark	histo	sad	bfs	spmv	stencil	lbm
FAIR	1.69	1.95	1.54	2.36	2.19	1.96
GREEDY	1.61	1.87	1.51	2.31	2.16	1.98
SMOOTH	1.64	1.97	1.53	2.33	2.17	1.95

Table 5.2: Normalized Performance Improvement over *NOMINAL*, Synthetic BE Tasks

line, corresponding to a cluster interval, stores the initial ID for the interval, as well as the number of remaining blocks in all successive intervals. At run-time, the process then matches the IDlist written by the most recently started thread block with the interval table to determine the number of not-yet-completed thread blocks for each cluster; note this is possible because blocks are activated in ID order, hence, if block ID i is executing, then we know that all blocks with ID less than i must either be executing (hence in IDlist) or have finished.

5.4 Evaluation

We evaluate the performance of our dynamic budget allocation approach based on the same benchmarks from Parboil used in Section 4.5. We compare the *FAIR, GREEDY* and *SMOOTH* allocation schemes against the *NOMINAL* allocation, where the same nominal budget is used for all regulation periods of a given kernel. As discussed in Section 3.1, we determine the nominal budget for kernel κ as the maximum $\overline{Q}$ such that the slowdown $\big(G^e(\overline{Q},\{N_i\},0) - G^e(0,\{N_i\},0)\big)/G^e(0,\{N_i\},0)$ is equal to 10%; the obtained values are listed in Table 5.1. Under *NOMINAL*, the kernel is run without instrumentation and no extra CPU process. For *FAIR, GREEDY* and *SMOOTH*, as discussed in Section 5.3 we need to instrument each kernel, and run a user-level CPU process to perform the on-line budget computation. We experimentally determined that the process can cause at most 470 LLC misses during each regulation period; since these misses cause extra interference to the GPU, we have to adjust the budget assigned to BE cores for the dynamic schemes by subtracting such LLC amount for every regulation period. This leads to a lower adjusted

Benchmark	histo	sad	bfs	spmv	stencil	lbm
FAIR, 462.libquantum	2.05	2.98	1.78	2.65	2.43	2.21
GREEDY, 462.libquantum	2.09	2.89	1.75	2.66	2.46	2.2
SMOOTH, 462.libquantum	2.04	2.93	1.76	2.67	2.42	2.19
FAIR, 403.gcc	1.36	1.42	1.23	1.35	1.27	1.29
GREEDY, 403.gcc	1.35	1.39	1.21	1.37	1.29	1.26
SMOOTH, 403.gcc	1.37	1.37	1.22	1.39	1.25	1.27
FAIR, 458.sjeng	1.14	1.11	1.07	1	1	1.03
GREEDY, 458.sjeng	1.11	1.12	1.06	1	1	1.04
SMOOTH, 458.sjeng	1.13	1.13	1.07	1	1	1.05
FAIR, Q mean (%)	41.77	49.1	36.3	74.4	70.4	59.6
GREEDY, Q mean (%)	42.02	48.9	36.2	74.4	70.6	59.6
SMOOTH, Q mean (%)	41.63	49	36.2	74.6	70.6	59.4
FAIR, Q std (%)	12.84	20.04	9.67	21.11	20.69	14.70
GREEDY, Q std (%)	20.09	13.88	9.67	13.82	10.81	11.28
SMOOTH, Q std (%)	8.35	8.16	3.33	13.43	9.5	6.52

Table 5.3: Performance Results, SPEC BE Tasks

nominal budget for the first regulation period of each kernel, see Table 5.1. As for the execution time of the budget-computation process, we measured a worst-case time of 10us to compute $\{R_i\}$, and 10us to execute Algorithm 3. Also accounting for the time to communicate with the BWLOCK++ kernel module, we configured the process to start 50us before the beginning of each regulation period.

Figure 5.1 provides graphs for all benchmarks, where the BE cores execute the synthetic bandwidth benchmark. Specifically, we report results in terms of assigned budget ratio Q for each allocation scheme and regulation period over a single run of each kernel. For the same scenario, Table 5.2 shows the performance improvement for each dynamic scheme, in terms of memory requests issued by the BE cores during the execution of the kernel, averaged over a million runs and normalized based on *NOMINAL*. As we can see, the performance improvement of the three schemes is similar, but they behave very differently in terms of budget allocation over time, with *SMOOTH* achieving by far the most uniform distribution. We also note that the performance improvement is significant, ranging from 51% to 136%.

Finally, we repeat the same experiments while running a benchmark from the SPEC2006 suite [36] on each BE core. We selected three benchmarks with different memory intensive-

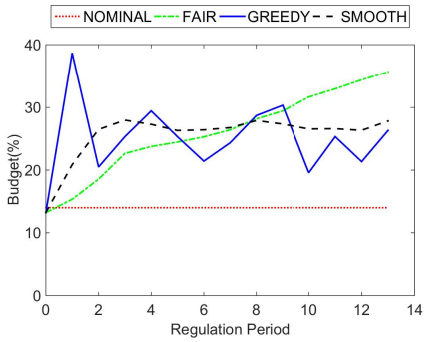
ness [74]: 462.libquantum (high), 403.gcc (medium), and 458.sjeng (low). Table 5.3 represents the results in terms of performance improvement over *NOMINAL* for each SPEC benchmark, as well as mean and standard deviation of the assigned budget ratio Q over all regulation periods. We find that for 462.libquantum, the performance improvement is higher compared to the synthetic benchmarks; this is because the other two benchmarks stress the memory less, thus resulting in less interference and more slack for the GPU kernel. The three dynamic allocation schemes again perform similarly, with the exception of the standard deviation, which is significantly smaller for *SMOOTH*.

5.5 Discussion and Conclusions

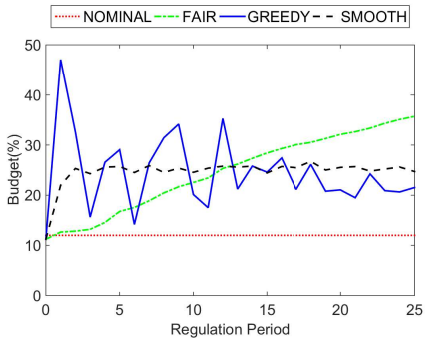
We have shown how the WCET estimation framework of Chapter 4 can be used to dynamically adjust the budget assigned to best-effort cores at run-time. To reduce run-time overhead, we introduce a new Algorithm 3 that performs the budget computation in linear time in the number $\mathcal{C}$ of thread block clusters. We also show how to optimize the computation of the remaining thread blocks at run-time based on information provided through kernel instrumentation. Results show that our implementation achieves low overhead while leading to significant performance improvements for best-effort tasks.

Based on our evaluation, we conclude that the three discussed allocation schemes, namely *FAIR, GREEDY* and *SMOOTH*, lead to similar average improvements in terms of throughput of best-effort applications. However, among those, *SMOOTH* shows by far the most stable budget allocation over time. While in our evaluation we ran a single benchmark per BE core, in practice we should expect that each BE core schedules a plurality of tasks in a time-sharing fashion; hence, we believe that *SMOOTH* can provide the fairest allocation of memory bandwidth over time to best-effort applications.

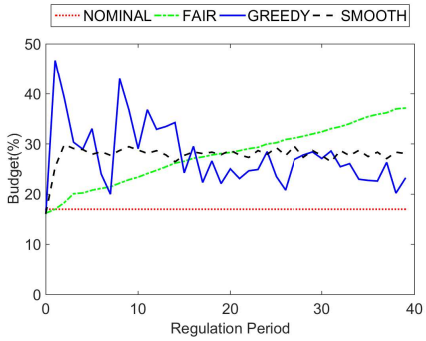
Finally, we note that the significant gains shown in Table 5.2 might appear surprising, since for this part of the evaluation we used the same memory-intensive benchmarks that we also employed to determine the WCET of thread blocks; hence, it might seem strange that there is a significant amount of slack in the kernel execution. As discussed in Section 4.5.4, we believe that this is both due to WCET overestimation, and because the benchmarks fail to cause the actual worst case, especially with $sync = 0$ in this scenario.



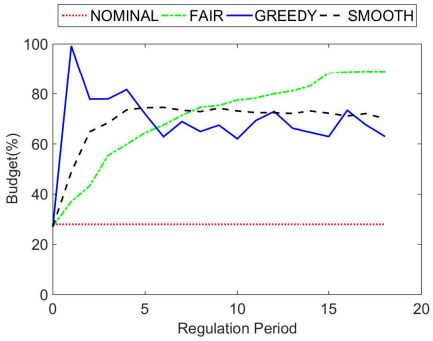
(a) histo



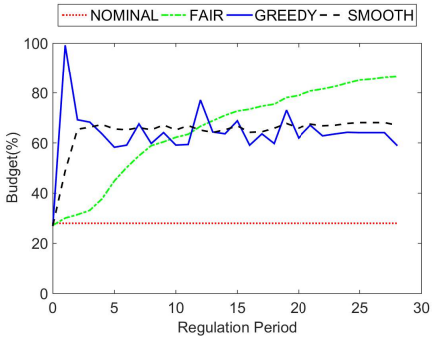
(b) sad



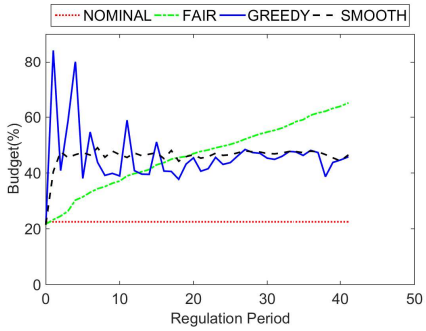
(c) bfs



(d) spmv



(e) stencil



(f) lbm

Figure 5.1: Budget Distribution over Time

Chapter 6

Conclusions and Future Work

Bounding interference effects is essential for the certification of multi-core real-time systems. Bandwidth throttling is an effective mechanism to protect real-time application from main memory interference, but it can lead to significant performance penalties for best-effort applications. To mitigate such performance impact, in this paper we have proposed a dynamic throttling scheme which adjusts the bandwidth budget assigned to best effort cores by exploiting the slack accumulated by a real-time GPU kernel. We proposed a methodology to estimate the progress of a GPU kernel at run-time. Our methodology is based on the observation that a kernel executes a large number of threads with the same code; while such code can include control instructions, the number of different program paths is usually limited. We thus classify groups of threads into clusters, each with different execution time profiles; then, at run-time, we count the number of completed groups for each cluster as a measure of progress. Following the proposed progress mechanism, we introduce a measurement-based WCET approach to estimate the execution time of a kernel based on its remaining number of thread groups per cluster, and the bandwidth threshold for BE cores. Using the discussed WCET estimation approach, we then show how to re-compute the BE bandwidth online while ensuring that the kernel completes within its original WCET. Our scheme significantly increases the throughput of memory-intensive, best-effort applications, up to 2.98x in our evaluation.

The presented work could be extended in several different directions. First of all, as noted in Section 3.1, each GPU segment comprises the execution of a kernel, but also memory copies that might be required to duplicate memory buffers held by the CPU and the GPU. Such copies are performed by hardware copy engines on the GPU, essentially specialized DMA units. Under BWLOCK++, both the memory copies and the kernel execution are protected by regulating memory accesses of best-effort cores; however, in

our current framework we only dynamically adjust the memory budget during the kernel execution. A key challenge in extending the approach to memory copies is that we do not have a simple way to estimate the progress of the copy engines; this would effectively require breaking each memory copy into multiple smaller operations, so that the completion of each operation can be tracked in software. Furthermore, at least for the tested benchmarks, the length of the memory copies tend to be quite small, in most case smaller than the duration of a regulation period, limiting our ability to recompute the budget.

The approach could also be extended to protect CPU segments from memory interference by either other CPU applications, or GPU kernels. In particular, we believe that hybrid CPU analysis [3, 18] can be adapted to estimate CPU progress in a way compatible with our framework. In this sense, one could investigate a more general system model comprising multiple CPUs and GPUs with any combinations of real-time processing elements. In such a system, we should protect CPUs, GPUs, or the combination of them from the effect of best-effort processing elements.

Finally, our approach relies on software memory regulation. Recent work [29] has shown that performing regulation in hardware can lead to a much finer regulation granularity and significant improvements in predictability for real-time applications. Hence, it would be interesting to develop a combined hardware-software approach to adjust at run-time the parameters of the hardware regulator.

References

[1] BWLOCK++ Github Repository. https://github.com/wali-ku/BWLOCK-GPU/tree/master/kernel_module.

[2] ISO, ISO 26262 road vehicles – functional safety. 2011.

[3] RapiTime. https:https://www.rapitasystems.com/products/rapitime, 2013.

[4] Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. Dynamic memory bandwidth allocation for real-time gpu-based soc platforms. In *2020 International Conference on Embedded Software*, 2020.

[5] Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software*, pages 1–10, 2014.

[6] Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[7] Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 285–296. IEEE, 2015.

[8] Waqar Ali and Heechul Yun. Work-in-progress: Protecting real-time gpu applications on integrated cpu-gpu soc platforms. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 141–144. IEEE, 2017.

[9] Waqar Ali and Heechul Yun. Protecting real-time gpu kernels on integrated cpu-gpu soc platforms. In *30th EUROMICRO Conference on Real-Time Systems (ECRTS'18)*, pages 3:1–3:2, 2018.

[10] Yannick Allard, Geoffrey Nelissen, Joel Goossens, and Dragomir Milojevic. A context aware cache controller to bridge the gap between theory and practice in real-time systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.

[11] ARM Inc. ARM Cortex-A57 MPCore Processor Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488c/BIIBJJGG.html.

[12] Michael Garrett Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2019.

[13] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24. IEEE, 2016.

[14] Kostiantyn Berezovskyi, Konstantinos Bletsas, and Björn Andersson. Makespan computation for gpu threads running on a single streaming multiprocessor. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 277–286. IEEE, 2012.

[15] Kostiantyn Berezovskyi, Fabrice Guet, Luca Santinelli, Konstantinos Bletsas, and Eduardo Tovar. Measurement-based probabilistic timing analysis for graphics processor units. In *International Conference on Architecture of Computing Systems*, pages 223–236. Springer, 2016.

[16] Guillem Bernat, Antoine Colin, and Stefan M Petters. Wcet analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 279–288. IEEE, 2002.

[17] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems*, 20(4):553–566, 2009.

[18] Adam Betts. *Hybrid Measurement-Based WCET Analysis using Instrumentation Point Graphs*. PhD thesis, Citeseer, 2008.

[19] Adam Betts and Alastair Donaldson. Estimating the wcet of gpu-accelerated applications using hybrid analysis. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 193–202. IEEE, 2013.

[20] Paolo Burgio, Andrea Marongiu, Paolo Valente, and Marko Bertogna. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, pages 1–8. IEEE, 2015.

[21] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for gpu with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130. IEEE, 2018.

[22] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. Sigamma: Server based integrated gpu arbitration mechanism for memory accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 48–57. ACM, 2017.

[23] Trishul M Chilimbi, Mark D Hill, and James R Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, 2000.

[24] Leonardo Milhomem Franco Christino and Fernando dos Santos Osório. Gpu-services: Real-time processing of 3d point clouds for robotic systems using gpus. In *2015 12th Latin American Robotics Symposium and 2015 3rd Brazilian Symposium on Robotics (LARS-SBR)*, pages 151–156. IEEE, 2015.

[25] Yadolah Dodge. *The concise encyclopedia of statistics*. Springer Science & Business Media, 2008.

[26] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. 2014.

[27] Glenn A Elliott, Bryan C Ward, and James H Anderson. Gpusync: A framework for real-time gpu management. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 33–44. IEEE, 2013.

[28] Glenn A Elliott, Kecheng Yang, and James H Anderson. Supporting real-time computer vision workloads using openvx on multicore+ gpu platforms. In *2015 IEEE Real-Time Systems Symposium*, pages 273–284. IEEE, 2015.

[29] Farzad Farshchi, Qijing Huang, and Heechul Yun. Bru: Bandwidth regulation unit for real-time multicore processors. In *Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2020*. IEEE, 2020.

[30] Björn Forsberg, Luca Benini, and Andrea Marongiu. Heprem: Enabling predictable gpu execution on heterogeneous soc. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 539–544. IEEE, 2018.

[31] Björn Forsberg, Andrea Marongiu, and Luca Benini. Gpuguard: Towards supporting a predictable execution model for heterogeneous soc. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 318–321. European Design and Automation Association, 2017.

[32] Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):53, 2018.

[33] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium*, pages 68–77. IEEE, 2009.

[34] Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246. IEEE, 2017.

[35] Mohamed Hassan and Rodolfo Pellizzoni. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, 2018.

[36] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[37] Seyedmehdi Hosseinimotlagh and Hyoseung Kim. Thermal-aware servers for real-time tasks on multi-core gpu-integrated embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 254–266. IEEE, 2019.

[38] Yijie Huangfu and Wei Zhang. Static wcet analysis of gpus with predictable warp scheduling. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 101–108. IEEE, 2017.

[39] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41. IEEE, 2019.

[40] John L. Hennessy, David A. Patterson. A New Golden Age for Computer Architecture. https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext, 2019.

[41] Leslie A Johnson et al. Do-178b, software considerations in airborne systems and equipment certification. *Crosstalk, October*, 199, 1998.

[42] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 57–66. IEEE, 2011.

[43] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[44] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154. IEEE, 2014.

[45] Hyoseung Kim, Pratyush Patel, Shige Wang, and Ragunathan Raj Rajkumar. A server-based approach for predictable gpu access control. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2017.

[46] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems*, 48(6):638–680, 2012.

[47] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224. IEEE, 1997.

[48] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, 2013.

[49] Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu. Combining prem compilation and ilp scheduling for high-performance and predictable mpsoc execution. In *Proceedings of the 9th International Workshop*

on Programming Models and Applications for Multicores and Manycores, pages 11–20, 2018.

[50] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 87–96, 2015.

[51] Frank Mueller. Compiler support for software-based cache partitioning. In *ACM Sigplan Notices*, volume 30, pages 125–133. ACM, 1995.

[52] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H Anderson, F Donelson Smith, Alex Berg, and Shige Wang. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 353–364. IEEE, 2017.

[53] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011.

[54] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746. IEEE, 2010.

[55] Arthur Pyka, Mathias Rohde, and Sascha Uhrig. Extended performance analysis of the time predictable on-demand coherent data cache for multi-and many-core systems. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 107–114. IEEE, 2014.

[56] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):164, 2017.

[57] Muhammad Husni Santriaji and Henry Hoffmann. Merlot: Architectural support for energy-efficient real-time processing in gpus. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 214–226. IEEE, 2018.

[58] Nivedita Sritharan, Anirudh M Kaushik, Mohamed Hassan, and Hiren D Patel. Hourglass: Predictable time-based cache coherence protocol for dual-critical multi-core systems. *CoRR, abs/1706.07568*, 2017.

[59] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[60] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois at Urbana-Champaign, 2012.

[61] Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A real-time scratchpad-centric os for multicore embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 2016.

[62] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared l2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33, 2007.

[63] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.

[64] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Addressing isolation challenges of non-blocking caches for multicore real-time systems. *Real-Time Systems*, 53(5):673–708, 2017.

[65] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167. IEEE, 2013.

[66] Saud Wasly and Rodolfo Pellizzoni. Hiding memory latency using fixed priority scheduling. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86. IEEE, 2014.

[67] Simon Wegener. Towards multicore wcet analysis. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[68] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. *ACM SIGPLAN Notices*, 52(4):483–496, 2017.

[69] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[70] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89. IEEE, 2008.

[71] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, 2012.

[72] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751, 2015.

[73] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 184–195. IEEE, 2015.

[74] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

[75] Wei Zhang and Jun Yan. Static timing analysis of shared caches for multicore processors. *Journal of Computing Science and Engineering*, 6(4):267–278, 2012.

[76] Husheng Zhou, Soroush Bateni, and Cong Liu. Sˆ 3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 190–201. IEEE, 2018.

[77] Husheng Zhou, Guangmo Tong, and Cong Liu. Gpes: A preemptive execution system for gpgpu computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97. IEEE, 2015.

Appendix

Instrumentation Code: Measuring Thread Block Execution

Listing 1 shows the CUDA code used to measure the execution time of each thread block, as detailed in Section 4.4. Note that compared to Algorithm 1, the presented code also extracts the execution time for the last M thread blocks of the kernel. `M` and `Nk` represent the number M of thread blocks that are executed simultaneously and the total number of thread blocks N_κ, respectively. As discussed in Section 4.4, we allocate two main data structures. `ExecTime` is used by the GPU to store the computed execution time of each thread block; while `TimeAr` is an array of type `Block`, where each `Block` element is a structure comprising the `ID` and the start time of an active thread block (`clk`). Finally, `CurrThreadBlocks` is a temporary array used to save the current running thread blocks' `ID`s.

The code in `Kernel_Function` represents the code added at the beginning of the GPU kernel κ under analysis. At the beginning of the kernel function for the first thread of a thread block, we measure the time to be the start for that thread block. If the block is among the first M blocks, then no thread block is terminated, and we only push its `ID` and start time to the `TimeAr` array. If the new thread block is not among the first M thread blocks, then we call the `Compute_ExecTime` function to determine which thread block is terminated and calculate its execution time (Lines 4-7 and 10-11 of Algorithm 1). As discussed in Section 4.4, we still need to measure the execution time of the last M thread blocks. We use the start time of the last thread of the final M thread blocks to be their finish time; using the finish time and start time of last M thread blocks, we can estimate their execution times.

Listing 1: Thread Block Execution Time Measurement

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <stdint.h>
#include <cooperative_groups.h>

struct Block{
    int ID;
    clock_t clk;
};
const int M = 8;
const int Nk = Number_of_Thread_Blocks;
volatile __device__ Block TimeAr[M];
volatile __device__ int CurrThreadBlocks[M];

__device__ clock_t get_clock(void)
{
    clock_t clock;
    asm("mov.u32␣%0,␣%clock;" : "=r"(clock) );
    return clock;
}

__device__ void Compute_ExecTime(clock_t clk, int blockID, double*
   ExecTime)
{
    for(int i=0;i<M;i++)
        Asm("declare␣i32␣@llvm.nvvm.read.ptx.sreg.ctaid.x(i)" : "=r"(
           CurrThreadBlocks[i]));
    for(int index=0;index<M;index++)
    {
        bool flag = 0;
        for(int j=0;j<M;j++)
            if(CurrThreadBlocks[j]==TimeAr[index].ID)
            {
                flag = 1;
                break;
            }
        if(flag==0)
        {
            ExecTime[TimeAr[index].ID] = (double)(clk-TimeAr[index].clk)/
               CLOCKS_PER_SEC;
            TimeAr[index].clk = clk;
            TimeAr[index].ID = blockID;
            return;
        }
```

```
    }
}

__global__ void Kernel_Function(double * ExecTime)
{
    if(threadIdx.x==0)
    {
        clock_t clk = get_clock();
        if(blockIdx.x<M)
        {
            TimeAr[blockIdx.x].clk = clk;
            TimeAr[blockIdx.x].ID = blockIdx.x;
        }
        else
            Compute_ExecTime(clk, blockIdx.x,ExecTime);
    }
    else if ((Nk-blockIdx.x)<=M && threadIdx.x==MAX_THREADS_PER_BLOCK-1)
    {
        clock_t clk = get_clock();
        int index;
        for(index=0;index<M;index++)
        {
            if(blockIdx.x==TimeAr[index].ID)
                break;
        }
        ExecTime[blockIdx.x] = (double)(clk-TimeAr[index].clk)/
           CLOCKS_PER_SEC;
    }

        .
        .
        .
}

int main()
{
        double * ExecTime;
        cudaMallocManaged(&ExecTime, Nk*sizeof(double) );
        Kernel_Function(ExecTime);
        .
        .
        .
}
```

Instrumentation Code: Run-time Budget Allocation

Listing 2 shows the CUDA code for the kernel instrumentation required to run the online budget allocation, as detailed in Section 5.3. We allocate a data structure named `CurrThreadBlocks` in main memory that can be accessed by both the GPU and the user-level process. We instrument the kernel code so that it writes the current clock value at the beginning of the first thread block. We write the start time of the kernel in the memory using `StartKernel` for the synchronization; hence the real-time CPU, which runs the process for budget allocation in parallel with the kernel, knows when the kernel starts the execution. We used the `start` flag to make sure that we only write the start of the kernel at the beginning of the first thread block. We used `clock()` function which is a global timer shared with the CPU. `StartKernel` and `CurrThreadBlocks`are allocated in the main memory using cudaMallocManaged, and the user-space process can access their address for reading the values. Each successive thread block writes to the `CurrThreadBlocks` the IDlist of concurrent blocks, which is used by the budget-computation process to determine the remaining blocks $\{R_i\}$.

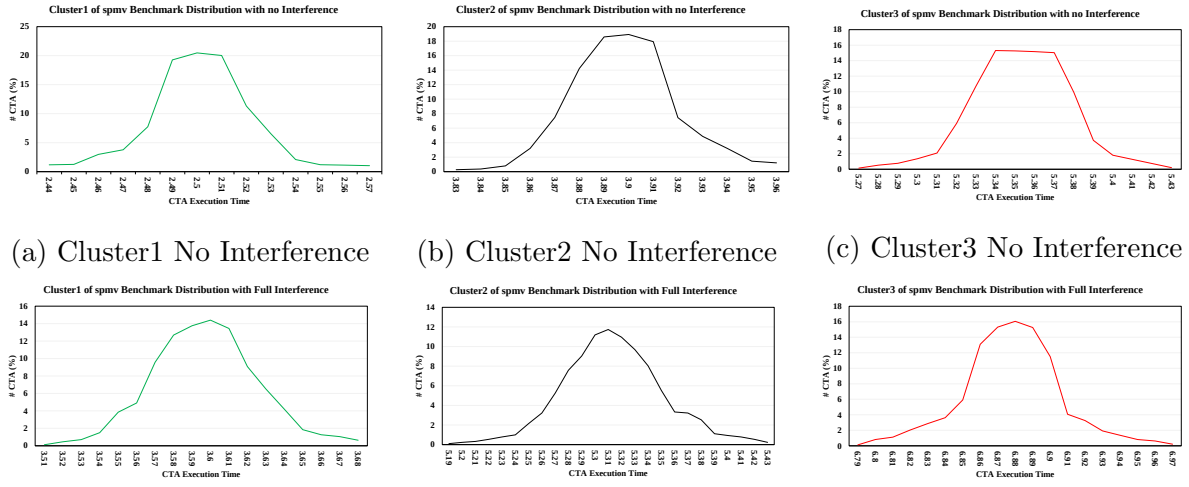
Listing 2: Run-Time Instrumentation

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <stdint.h>
#include <cooperative_groups.h>

const int M = 8; //Depending on the Kernel
volatile __device__ bool start = 1;

__global__ void Kernel_Function(int* CurrThreadBlocks,clock_t*
    StartKernel)
{
     if(threadIdx.x==0)
     {
          if(start==1)
          {
                   (*StartKernel) = clock();
                   start = 0;
          }
          for(int i=0;i<M;i++)
                   Asm("declare␣i32␣@llvm.nvvm.read.ptx.sreg.ctaid.x(i)" : "
                      =r"(CurrThreadBlocks[i]));
          }
          .
```

```
        .
}

int main()
{
        int *CurrThreadBlocks;
        cudaMallocManaged(&CurrThreadBlocks, M*sizeof(int) );
        clock_t* StartKernel;
        cudaMallocManaged(&StartKernel, sizeof(clock_t) );
        kernel(CurrThreadBlocks,StartKernel);
        .
        .
}
```

Measured EDFs of Parboil Benchmark

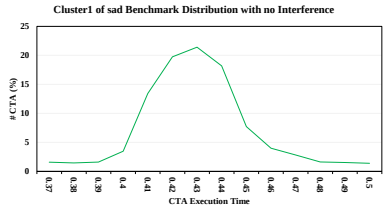
Figures 1, 2, 3, 4, and 5 show the Empirical Distribution Function (EDF) for the execution times of the thread block clusters of the spmv, sad, bfs, lbm, and stencil Parboil benchmarks, respectively. As in Figure 4.1, we show the distributions both for the case of no interference ($Q = 0$), which is used to derive e_i^0, and the case of full interference ($Q = 1$), which is used to derive e_i^1.



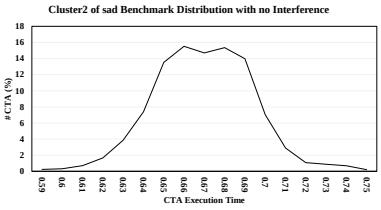
(a) Cluster1 No Interference (b) Cluster2 No Interference (c) Cluster3 No Interference

(d) Cluster1 Full Interference (e) Cluster2 Full Interference (f) Cluster3 Full Interference

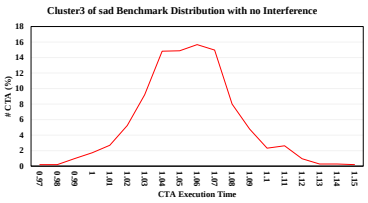
Figure 1: Distribution of block execution times for spmv benchmark



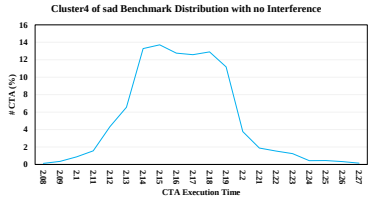
(a) Cluster1 No Interference



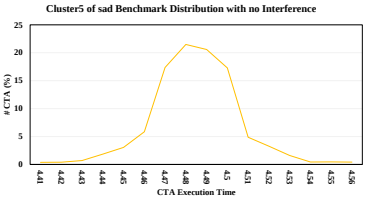
(b) Cluster2 No Interference



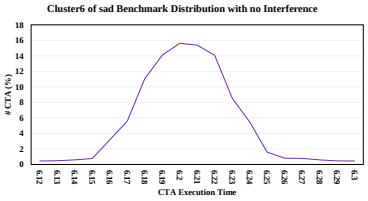
(c) Cluster3 No Interference



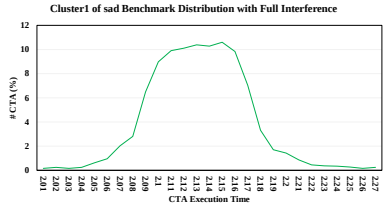
(d) Cluster4 No Interference



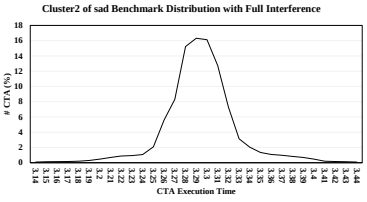
(e) Cluster5 No Interference



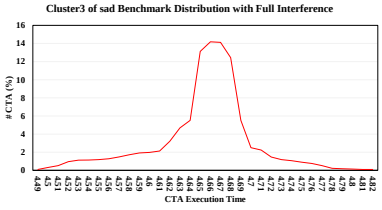
(f) Cluster6 No Interference



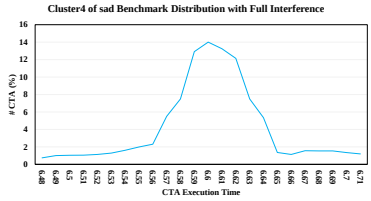
(g) Cluster1 Full Interference



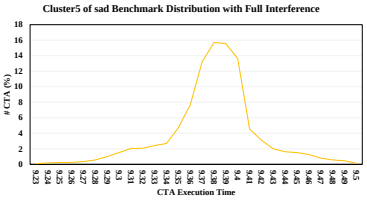
(h) Cluster2 Full Interference



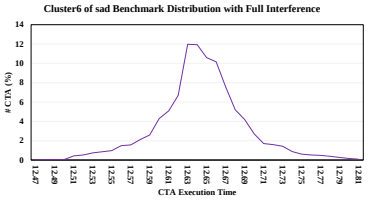
(i) Cluster3 Full Interference



(j) Cluster4 Full Interference

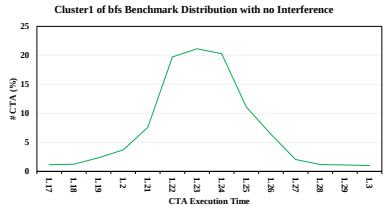


(k) Cluster5 Full Interference

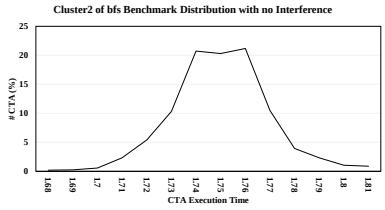


(l) Cluster6 Full Interference

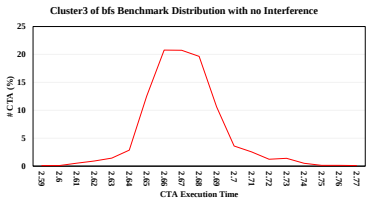
Figure 2: Distribution of block execution times for sad benchmark



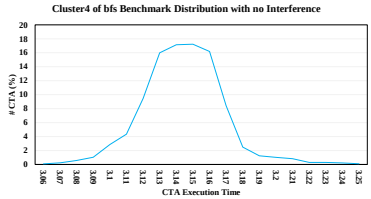
(a) Cluster1 No Interference



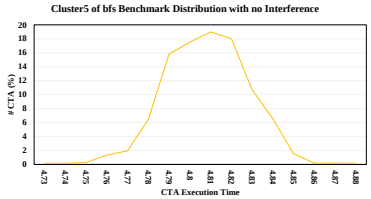
(b) Cluster2 No Interference



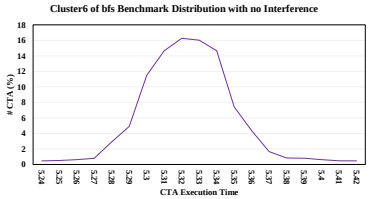
(c) Cluster3 No Interference



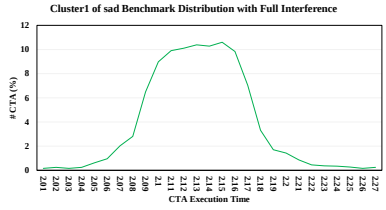
(d) Cluster4 No Interference



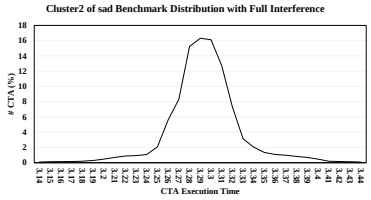
(e) Cluster5 No Interference



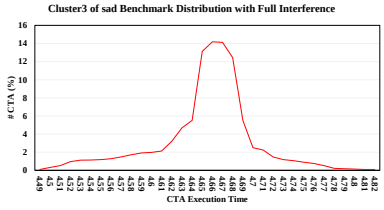
(f) Cluster6 No Interference



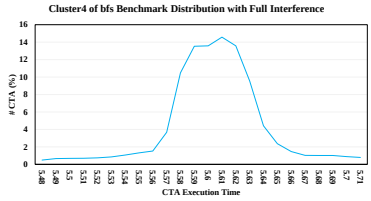
(g) Cluster1 Full Interference



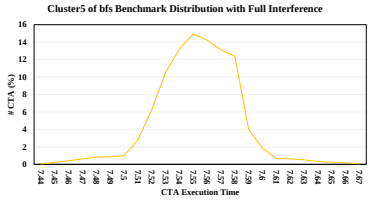
(h) Cluster2 Full Interference



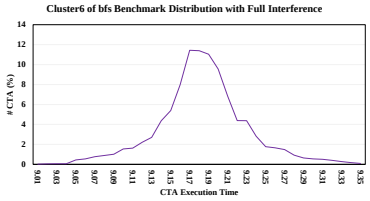
(i) Cluster3 Full Interference



(j) Cluster4 Full Interference

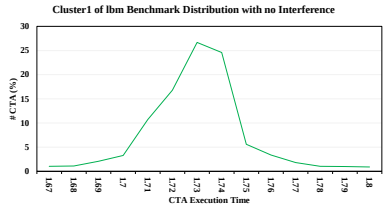


(k) Cluster5 Full Interference

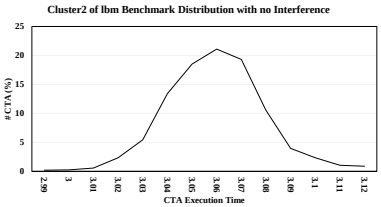


(l) Cluster6 Full Interference

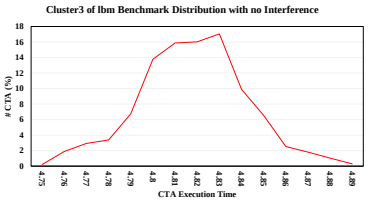
Figure 3: Distribution of block execution times for bfs benchmark



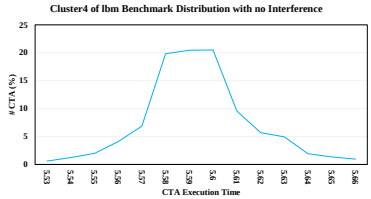
(a) Cluster1 No Interference



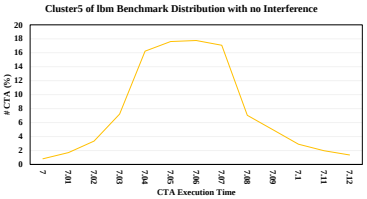
(b) Cluster2 No Interference



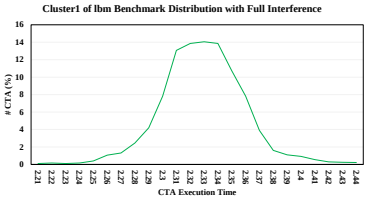
(c) Cluster3 No Interference



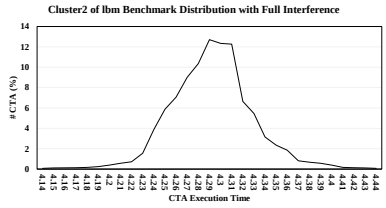
(d) Cluster4 No Interference



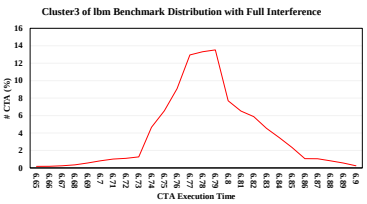
(e) Cluster5 No Interference



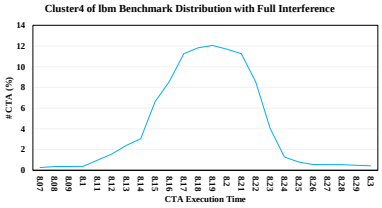
(f) Cluster1 Full Interference



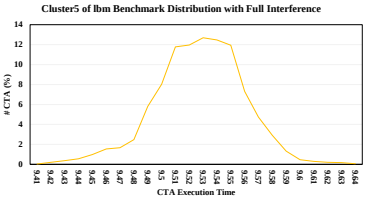
(g) Cluster2 Full Interference



(h) Cluster3 Full Interference

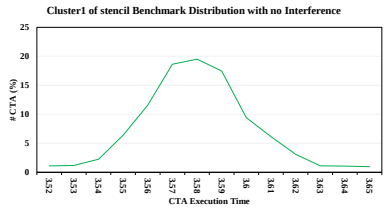


(i) Cluster4 Full Interference

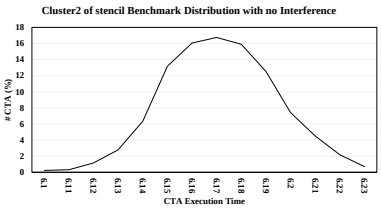


(j) Cluster5 Full Interference

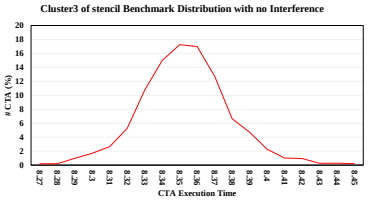
Figure 4: Distribution of block execution times for lbm benchmark



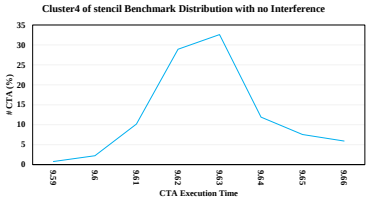
(a) Cluster1 No Interference



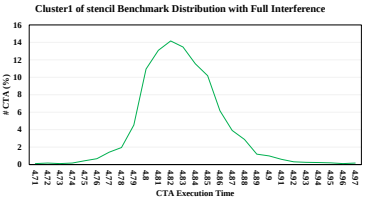
(b) Cluster2 No Interference



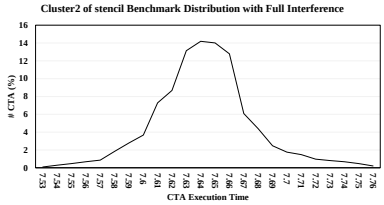
(c) Cluster3 No Interference



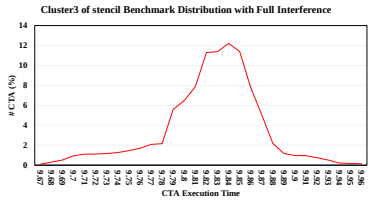
(d) Cluster4 No Interference



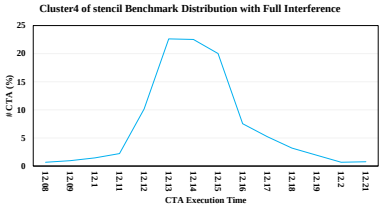
(e) Cluster1 Full Interference



(f) Cluster2 Full Interference



(g) Cluster3 Full Interference



(h) Cluster4 Full Interference

Figure 5: Distribution of block execution times for stencil benchmark